Implementierung einer skriptbasierten Steuerung von Simulationsprozessen in der Triebwerksauslegung zur Identifizierung optimaler Leistungsmerkmale

BACHELORARBEIT

für die Prüfung zum

Bachelor of Engineering

im Studiengang Informationstechnik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

MARVIN NÖTHEN

Bearbeitungszeitraum	24.06.2019 - 16.09.2019

Ehrenwörtliche Erklärung

gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Köln - Porz, den 13. September 2019

Kurzfassung

Der Triebwerksvorentwurf ist ein hoch iterativer Prozess, in dem verschiedenste Disziplinen eines Triebwerks betrachtet werden müssen. Als unterstützendes Werkzeug zur Durchführung des hoch iterativen Prozesses kann das Programmsystem Gas Turbine Laboratory (GTlab) eingesetzt werden. GTlab bietet die Möglichkeit Berechnungsverfahren in Prozessketten zu verschalten und diese über eine interne Prozesssteuerung auszuführen. Dabei werden die Komponenten einer Prozesskette nach einem hierarchischen Prinzip, sequenziell ausgeführt.

Ziel der vorliegenden Bachelorarbeit ist die Integration einer skriptbasierten Prozesssteuerung in GTlab. Dadurch soll die hierarchische Ausführung von Berechnungsverfahren aufgebrochen und die Flexibilität im Aufbau von Prozessketten erhöht werden. In einem initialen Schritt wurde das zentrale Datenmodell und die bestehende Prozesssteuerung von GTlab analysiert. Die daraus gewonnenen Erkenntnisse wurden zur Konzeptionierung der skriptbasierten Prozesssteuerung verwendet. Das entstandene Konzept konnte innerhalb eines neu erstellten GTlab-Moduls realisiert werden. Darin wurde die skriptbasierte Prozesssteuerung unter Verwendung der GTlab-Schnittstellen als ausführbare Komponente von GTlab implementiert. Nach dem Start der Komponente wird ein frei definierbares Python-Skript in einer Python-Umgebung ausgeführt. Die Python-Umgebung ist mit Methoden zur Konfiguration und Ausführung von GTlab-internen Berechnungsverfahren ausgestattet. Auf diese Weise ist das flexible Verschalten von Berechnungsverfahren und somit die Definition von Prozessketten mit Hilfe von Python-Code möglich. Das GTlab-Modul kann zukünftig dynamisch in GTlab integrieren werden, wodurch die skriptbasierte Prozesssteuerung verfügbar wird.

Abstract

The preliminary engine design is a highly iterative process in which a wide variety of engine disciplines have to be considered. The software framework Gas Turbine Laboratory (GTlab) can be used to support this highly iterative process. GTlab offers the possibility to interconnect calculation methods in process chains and to execute them via an internal process control environment. The components of a process chain are executed sequentially according to a hierarchical principle.

The aim of this bachelor thesis is the integration of a script-based process control environment into GTlab. This is intended to break up the hierarchical execution of calculation methods and increase flexibility in the structure of process chains. In an initial step, the central data model and the existing process control environment of GTlab were analyzed. The findings were for the development of a concept for the script-based process control environment. A new GTlab module was created in which the concept could be realized. By using the GTlab interfaces, the script-based process control environment was implemented as an executable component of GTlab. After starting the component, a user definable Python script is executed in a Python environment. The Python environment is equipped with methods for configuring and executing GTlab internal calculation methods. In this way the flexible interconnection of calculation methods and thus the definition of process chains with the help of Python code is possible. In future, the GTlab module can be dynamically integrated into GTlab, making script-based process control available.

Inhalt

Abkürzungsverzeichnis

DLR	**D**eutsches Zentrum für **L**uft- und **R**aumfahrt e.V.
AT-TWK	**A**ntriebs**t**echnik - **T**rieb**w**er**k**
GTlab	**G**as **T**urbine **Lab**oratory
UML	**U**nified **M**odeling **L**anguage

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Triebwerksvorauslegung stellt einen zentralen Aufgabenbereich der Abteilung Triebwerk (AT-TWK) am Institut für Antriebstechnik des Deutschen Zentrums für Luft- und Raumfahrt (DLR) dar. Dabei werden bestehende Triebwerke und Gastrubinen bewertet und zukünftige Antriebskonzepte entworfen. Als unterstützendes Werkzeug bei der Durchführung der Triebwerksvorauslegung, wird das Programmsystem Gas Turbine Laboratory (GTlab) federführend von der Abteilung AT-TWK entwickelt. GTlab ist eine interaktive, plattformübergreifende Simulations- und Vorentwurfsumgebung für Flugtriebwerke und Gasturbinen. Es ermöglicht die Durchführung von ersten thermodynamischen Berechnungen, bis hin zur Ermittlung dreidimensionaler Geometriedaten und deren Visualisierung. Zur Ausführung von Berechnungsprozessen verfügt GTlab über eine interne Prozesssteuerung. Diese erlaubt es verschiedenste Berechnungsverfahren zu verschalten und in Form von Prozessketten sequenziell auszuführen. Die sequenzielle Ausführung wird mit dieser Bachelorarbeit aufgebrochen, um die Flexibilität im Aufbau von Prozessketten zu erhöhen. Das Ziel ist die Integration einer skriptbasierten Prozesssteuerung in GTlab, die das Erstellen und Ausführen von Prozessketten in Form von Python-Skripten ermöglicht. Die skriptbasierte Prozesssteuerung wird dabei als Erweiterung neben der bisher bestehenden Prozesssteuerung integriert.

1.1 Motivation

Der Lebenszyklus eines Triebwerks, beginnend mit einem Vorentwurf bis hin zur Nutzung und der darauffolgenden Außerdienststellung, lässt sich in mehrere Phasen unterteilen. In der initialen Konzeptphase werden anhand weniger Parameter erste konzeptionelle Eigenschaften eines Triebwerks bestimmt. Dabei wird die Thermodynamik des Triebwerks betrachtet, die sich dazu verwenden lässt, konzeptionelle Geometrieabschätzungen einzelner Triebwerkskomponenten zu bestimmen. Die Ergebnisse der Konzeptphase

dienen als Einstieg in die darauffolgende Entwurfsphase und werden für die Erstellung eines Detailentwurfs benötigt. Im Verlauf des Detailentwurfs, wird die Genauigkeit der Geometriedaten unter Verwendung höherwertiger Simulationsprogramme stetig erhöht. Es folgt die Produktrealisierung, die als Grundstein für die Serienproduktion des Triebwerks dient. Daraufhin kann das Triebwerk, bis zu seiner Außerdienststellung, in Flugoperationen eingesetzt werden.

In Bezug auf den Lebenszyklus eines Triebwerks, lässt sich die vorliegende Arbeit innerhalb des Vorentwurfs der Konzeptphase einordnen. Diese beginnt mit ersten Vorstudien basierend auf Spezifikationen, die sich aus Marktanalysen und anderweitigen Anforderungen ergeben. Auf Grundlage der Vorstudien werden in einem hoch iterativen Prozess die verschiedenen Disziplinen eines Triebwerks betrachtet. Dabei werden Daten für Thermodynamik, Aerodynamik und Struktur des Triebwerkvorentwurfs ermittelt. Als unterstützendes Werkzeug bei der Durchführung des hoch iterativen Prozess, kann das Programmsystem GTlab verwendet werden. Es stellt Prozesselemente bereit, die dazu dienen, komponentenspezifische Berechnungsverfahren bezogen auf die einzelnen Disziplinen des Vorentwurfs auszuführen. Der Datenaustausch zwischen den Prozesselementen wird über ein zentrales Datenmodell vollzogen. Das bedeutet, dass ein Prozesselement beim Starten seine Eingabeparameter aus dem zentralen Datenmodell bezieht und die Ergebnisdaten in dieses zurückschreibt. Das zentrale Datenmodell bildet das System Triebwerk virtuell und standardisiert ab. Dazu verwaltet es für jede Triebwerkkomponente Daten, bezogen auf verschiedene Disziplinen und von unterschiedlicher Genauigkeit. Zudem bietet GTlab eine Infrastruktur zur Verteilung diverser Ressourcen, wie zum Beispiel Materialdaten. Um den Vorentwurfsprozess mit Hilfe von GTlab durchlaufen zu können, wird die interne Prozessteuerung benötigt. Diese erlaubt es Prozesselemente in Prozessketten hintereinander zu schalten und deren Ein- und Ausgabeparameter miteinander zu verknüpfen. Der Aufbau der Prozessketten folgt bislang einem fest vorgegebenen hierarchischem Prinzip. So werden beim Ausführen einer Prozesskette, die verwendeten Prozesselemente sequenziell ausgeführt.

Die sequenzielle Ausführung der Prozesselemente soll im Verlauf dieser Arbeit aufge-

brochen werden. Dazu soll es möglich werden, Prozessketten durch Python-Skripte zu definieren und unter Nutzung des in GTlab integrierten Python-Interpreters auszuführen. Die Prozesselemente müssen innerhalb der Python-Skripte instanziierbar, konfigurierbar und ausführbar sein. Zudem muss ein Zugriff auf das zentrale Datenmodell und die Ressourcenverwaltung ermöglicht werden. Der Vorteil, der durch das Erstellen von Prozessketten mittels Python-Skripten entstehen soll, besteht darin, dass die Ausführung von Prozesselementen flexibel definiert werden kann. So können Prozesselemente in Abhängigkeit von entstandenen Ergebnisdaten iterativ ausgeführt oder verzweigt werden.

1.2 Überblick

Die vorliegende Arbeit beginnt mit einer Einführung in die benötigten Grundlagen. Dazu wird die C++-Klassenbibliothek Qt vorgestellt, die ein fester Bestandteil der Architektur von GTlab ist. Sie erweitert die Programmiersprache C++ um zahlreiche Module und Klassen, die unter anderem die plattformübergreifende Programmierung von grafischen Benutzeroberflächen ermöglichen. Als weitere Grundlage wird die Klassenbibliothek PythonQt beschreiben, die die Vorteile der Qt-Klassenbibliothek in eine Python-Umgebung überführt und sie über einen Python-Interpreter nutzbar macht. Nach der grundlegenden Einführung folgt die Beschreibung der Aufgabenstellung, in der die Anforderungen an die vorliegende Arbeit spezifiziert werden. Zur Erfüllung der Aufgabenstellung wird die Arbeitsweise der Triebwerkvorentwurfsumgebung GTlab analysiert und beschrieben. Dabei werden insbesondere der Aufbau des zentralen Datenmodells und die Arbeitsschritte der Prozesssteuerung betrachtet. Die daraus gewonnen Erkenntnisse konnten genutzt werden, um eine skriptbasierte Prozesssteuerung in GTlab zu integrieren. Sie erlaubt es Berechnungsprozesses in GTlab über Python-Skripte zu definieren und ermöglicht dabei den Zugriff aufs zentrale Datenmodell. Die Erweiterungen die für die skriptbasierte Prozesssteuerung vorgenommen werden mussten, werden im Verlauf der Arbeit ausführlich präsentiert. Letztlich wird ein Anwendungsfall für die skriptbasierte Prozesssteuerung beschreiben, in dem die Abschätzung der Masse einer Triebwerkskomponente durchgeführt wird.

2 Grundlagen

Bei der Entwicklung des Programmsystems GTlab wird die Qt-Bibliothek verwendet. Sie stellt umfangreiche Funktionalitäten bereit, die als Erweiterung der Programmiersprache C++ genutzt werden können. In diesem Kapitel werden die Funktionalitäten der Qt-Bibliothek vorgestellt, die im Verlauf der vorliegenden Arbeit von Relevanz waren. Dazu zählt die hierarchische Verwaltung von Objekten, das Nutzen von Metainformation über Klassen und derer Instanzen und die Parallelisierung der Ausführung einzelner Programmteile. Zudem enthält das Kapitel eine Einführung in die PythonQt-Bibliothek, die es ermöglicht zur Laufzeit von C++-Anwendungen Python-Skripte an einen Python-Interpreter zu übergeben und über diesen die Funktionalitäten der Qt-Bibliothek zu nutzen. Zur Visualisierung der in dieser Arbeit angefertigten Implementierungen wurden Klassendiagramme verwendet, die durch *Unified Modeling Language* (UML) standardisiert sind. Die Bedeutung der Komponenten eines Klassendiagramms werden ebenfalls in diesem Kapitel beschreiben.

2.1 Die Qt-Bibliothek

Die Qt-Bibliothek ist eine Klassenbibliothek, die bei der Programmierung mit der Programmiersprache C++ eingesetzt werden kann. Sie erweitert die Programmiersprache C++ um zahlreiche Module und Klassen, die unter anderem die plattformübergreifende Programmierung von grafischen Benutzeroberflächen ermöglichen. Die Klassen der Qt-Bibliothek sind dadurch gekennzeichnet, dass ihre Klassennamen den Präfix *Q* besitzen. In vielen Programmteilen von GTlab wird auf die Klassen der Qt-Bibliothek zugegriffen. So wurde beispielsweise der hierarchische Aufbau des zentralen Datenmodells unter Verwendung der Klasse *QObject* realisiert. Die *QObject*-Klasse kann als Basisklasse der Qt-Bibliothek betrachtet werden, da die meisten von Qt bereitgestellten Klassen von der *QObject*-Klasse abgeleitet sind. Zur Hierarchisierung von Instanzen stellt die

QObject-Klasse eine Eltern-Kind-Beziehung bereit, über die sie sich selbst in Form eines Objektbaums organisieren kann. Das bedeutet, dass jeder *QObject*-Instanz ein Elternelement zugewiesen werden kann, welches ebenfalls von der Klasse *QObject* abstammen muss. Auf diese Weise können Instanzen der *QObject*-Klasse in beliebig breiten und beliebig tiefen Hierarchiebäumen verwalten werden. Dabei tragen die Elternobjekte die Verantwortung für die Kindobjekte. Wird ein Elternobjekt zerstört werden auch all seine Kindobjekte zerstört [1]. Neben der Hierarchisierung von Objekten wird die Qt-Bibliothek in GTlab zur dynamischen Erzeugung neuer Instanzen und zur Nutzung einer Introspektion eingesetzt. Durch die Introspektion können Informationen über Klassen und Objekte zur Laufzeit abgefragt werden. Die Qt-Bibliothek stellt dazu das Meta-Object-System (MOS) bereit. Alle Klassen die von der *QObject*-Klasse abgeleitet sind können im MOS registriert werden. Daraufhin ist ihnen eine Instanz der Klasse *QMetaObject* zugewiesen. In dieser sind Metainformationen wie beispielsweise Klassenname und Informationen über die Methoden der Klasse gespeichert. Mit Hilfe der *QMetaObject*-Instanz ermöglicht das MOS die Introspektion und das dynamische Erzeugen neuer Instanzen [2].

Im Triebwerksvorentwurf kommen Berechnungsverfahren von unterschiedlicher Komplexität zum Einsatz. Die Berechnungsverfahren können in GTlab integriert und innerhalb des Programmsystems ausgeführt werden. Bei besonders komplexen Berechnungen kann es zu langen Rechenzeiten kommen. GTlab ist so implementiert, dass es verschiedene Aufgaben, wie beispielsweise das Ausführen von Berechnungen und das Visualisieren von Daten, parallel ausführt. Dazu wurde eine Schnittstelle der Qt-Bibliothek verwendet, die als Werkzeug zur Implementierung der parallelen Ausführung genutzt werden kann. Um die Funktionsweise der Schnittstelle erläutern zu können, wird zunächst die parallele Ausführung von Programmteilen im Allgemeinen skizziert.

Die Ausführung eines Programms ist als ein Prozess zu betrachten. Es ist möglich die Ausführung bestimmter Programmteile innerhalb eines Prozesses auf unterschiedliche Ausführungsfäden zu verteilen. Solche Ausführungsfäden werden in der Informatik als Threads bezeichnet. Jeder Prozess besitzt mindestens einen Thread, der im Weiteren als Hauptthread bezeichnet wird. Im Hauptthread werden alle Aufgaben des Programms in

einer vorgegebenen Reihenfolge abgearbeitet. Programmteile, die sich als eigenständige Aufgabe isolieren lassen, können in einen eigenen Thread ausgelagert und somit parallel zum Hauptthread und zu den weiteren Threads eines Prozesses ausgeführt werden. Auf dies Weise lassen sich die Ausführungsgeschwindigkeit einzelner Aufgaben erhöhen und die gegenseitige Beeinträchtigung verschiedener Programmteilen verhindern [3]. Dabei ist zu beachten, dass bei der Programmierung von mehreren Threads Gefahren beim Zugriff auf gemeinsam genutzte Speicherressourcen auftreten können. Zudem stellt das Erstellen und Löschen von Threads einen hohen programmtechnischen Aufwand dar. Der Programmierer muss also abwägen, ob das Auslagern von Programmteilen in einen eigenen Thread für den entsprechenden Anwendungsfall sinnvoll ist. Die Qt-Bibliothek stellt mehrere Möglichkeiten bereit, solch eine Auslagerung durchzuführen. Eine davon ist durch die Klasse *QThreadPool* und die Schnittstelle *QRunnable* realisiert. Die *QThreadPool*-Klasse ermöglicht es auf eine Sammlung von wiederverwendbaren Threads zuzugreifen. Die Anzahl der Threads, die sich in der Sammlung befinden, ist auf die Anzahl der verfügbaren Prozessorkerne abgestimmt. Der Vorteil der Sammlung besteht darin, dass die Threads nicht durch den Programmierer erstellt und gelöscht werden müssen. Sie verbleiben in der Sammlung und können beliebig oft wiederverwendet werden. Die *QRunnable*-Schnittstelle stellt eine abstrakte Klasse dar. Klassen die von *QRunnable* abgeleitet sind und die entsprechenden Funktionalitäten implementieren können an die *QThreadPool*-Klasse übergeben werden. Die *QThreadPool*-Klasse führt die Implementierung der *QRunnable*-Unterklasse in einem eigenen Thread aus [4]. Weiter Möglichkeiten der Threadprogrammierung, die die Qt-Bibliothek bereitstellt, werden an dieser Stelle nicht beschreiben, da sie für die vorliegende Arbeit von geringer Bedeutung sind.

2.2 Die PythonQt-Bibliothek

PythonQt ist eine Programmbibliothek, die in der Programmiersprache C++ und unter Verwendung der Qt-Bibliothek geschrieben wurde. Sie ist darauf ausgelegt die Qt-Funktionalitäten in eine Python-Umgebung zu überführen und die Python-Umgebung in

eine bestehende C++-Anwendung zu integrieren. Dazu verwendet PythonQt die Python-Schnittstellen, die das Einbetten von Python in eine C++-Anwendung ermöglichen. Die Einbettung zielt darauf ab, die C++-Anwendung so zu erweitern, dass sie zur Laufzeit Python-Skripte an einem Python-Interpreter übergeben und ausführen lassen kann [5].

Skripte stellen kleine Programme dar, dessen Implementierungen meist innerhalb einer einzigen Quellcodedatei definiert sind. Python ist auf Grund seiner einfachgehaltenen Syntax, einer dynamischen Typisierung von Variablen und der Ausführung von Quellcode über einen Interpreter besonders zur Skriptprogrammierung geeignet [6]. Als Interpreter bezeichnet man ein Programm, welches Programmcode einlesen, übersetzen und ausführen kann. Im Gegensatz zu einem Compiler, der zum Beispiel bei den Programmiersprachen C und C++ eingesetzt wird, erzeugt der Interpreter keine speicherbaren Maschinencodedateien. Der Python-Interpreter Übersetzt den Quellcode zur Laufzeit in Bytecode und führt diesen in einer virtuellen Maschine aus. Auf Grund der Übersetzung, die der Interpreter bei jeder Ausführung erneut vornehmen muss, verlangsamt sich die Ausführungsgeschwindigkeit eines interpretierten gegenüber einem kompilierten Programm. Jedoch ermöglicht der Interpreter durch das Wegfallen des Kompilierungsschritts eine schnelle Entwicklung von kleineren Programmen und steigert die Flexibilität. So kann beispielsweise Python-Code geschrieben und unter Nutzung des Python-Interpreters sofort ausgeführt und getestet werden [7]. Durch die Einbettung von Python in eine C++ Anwendung, kann diese zur Laufzeit über Python-Skripte gesteuert und an individuelle Bedürfnisse angepasst werden. Um PythonQt einbinden zu können, muss die C++-Anwendung unter Verwendung der Qt-Bibliothek erstellt worden sein. PythonQt nutzt das MOS der Qt-Bibliothek, um Objekte der Klasse *QObject* in die Python-Umgebung zu überführen. Daraufhin kann auf alle Kindelemente und alle Funktionen des entsprechenden Objekts, die im MOS verfügbar sind, zugegriffen werden [8].

2.3 Unified Modeling Language

Bei der Durchführung der vorliegenden Arbeit wurde objektorientierter C++-Programmcode analysiert und erstellt. Als Hilfsmittel zur Visualisierung von Beziehungen zwischen Klassen wurde Unified Modeling Language (UML) verwendet. UML ist eine universell einsetzbare, visuelle Modellierungssprache, die unter anderem dazu verwendetet werden kann Softwaresysteme zu entwerfen, deren Aufbau zu visualisieren oder deren programmtechnischen Ablauf zu verdeutlichen [9]. Dazu standardisierte UML verschiedene Diagrammtypen und deren Komponenten. Die Semantik der UML-Diagramme ist an die objektorientierte Programmierung angelehnt [10]. In diesem Kapitel werden UML-Klassendiagramme erläutert. Dabei werden die Komponenten eines Klassendiagramms beschreiben, die im Verlauf der vorliegenden Arbeit verwendet wurden. Zudem werden Eigenarten der C++-spezifischen objektorientierten Programmierung erläutert, die für das Verständnis der Diagramme von Bedeutung sind.

Klassendiagrammen visualisieren die allgemeine Struktur eines Programmsystems. Dazu ist es möglich das Verhalten einzelner Klassen, sowie die Beziehung zwischen verschiedenen Klassen darzustellen. Eine Klasse wird im Klassendiagramm als Rechteck dargestellt, welches durch horizontale Linien in drei Bereiche unterteilt werden kann (sieh Abbildung 2.1). Im oberen Bereich wird der Klassenname platziert. Der mittlere Bereich dient zur Auflistung der Attribute und der untere Bereich zur Auflistung der Methoden einer Klasse. Bei der Darstellung einer Klasse ist es meist hilfreich nur die Attribute und

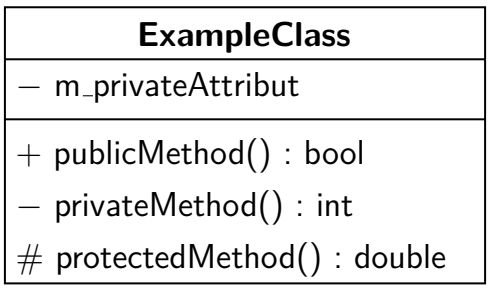

Abbildung 2.1
Darstellung einer Klasse im UML-Klassendiagramm

Methoden aufzulisten, die in dem entsprechenden Kontext betrachtet werden. Durch die vorangestellten Zeichen (+, –, #) lassen sich die Zugriffsrechte auf Attributen und Methoden darstellen. Das Pluszeichen bedeutet, dass das jeweilige Element öffentlich ist und somit auch außerhalb verwendet werden kann. Im Gegensatz dazu sind die privaten Attribute und Methoden, die nur innerhalb der Klassendefinition aufrufbar sind, mit dem Minuszeichen gekennzeichnet. Das Doppelkreuz bedeutet, dass das Attribut oder die Methode geschützt ist. Geschütze Klassenelemente können unter Verwendung des objektorientierten Prinzips der Vererbung innerhalb von Unterklassen aufgerufen werden [10]. Die Vererbung wird im UML-Klassendiagramm durch eine Generalisierungsbeziehung dargestellt, die im Folgenden näher beschrieben wird. Zudem wird die Darstellung von Assoziationen und Abhängigkeiten zwischen Klassen erläutert.

Generalisierung

Das objektorientierte Prinzip der Vererbung wird in einem UML-Klassendiagramm durch die Generalisierung repräsentiert. Die Generalisierung wird in Form einer durchgezogenen Linie und einem nicht eingefärbtem Dreieck als Verweis auf die Basisklasse dargestellt (sieh Abbildung 2.2) [10].

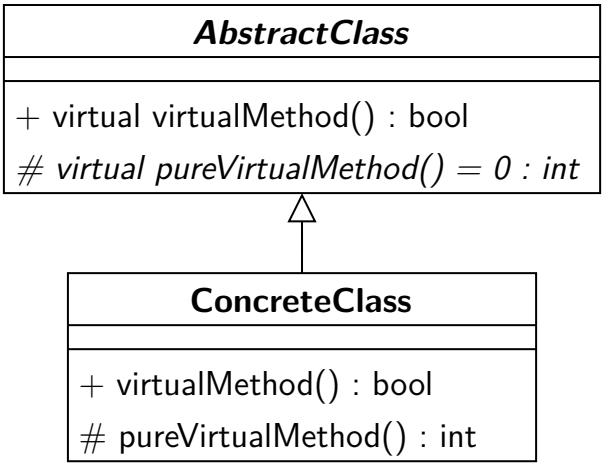

Abbildung 2.2
Generalisierung im UML-Klassendiagramm

Um die Bedeutung der Generalisierung zu verdeutlichen, muss das Prinzip der Vererbung genauer betrachtet werden. Die Vererbung ermöglicht das Erstellen von Basisklassen, in denen das allgemeine Verhalten von Objektfamilien definiert werden kann. Unterklassen können das Verhalten von Basisklassen erben und um zusätzliche Funktionalitäten erweitern. Zudem können Methoden aus einer Basisklasse in einer Unterklasse mit einem neuen Verhalten überschrieben werden. In der Programmiersprache C++ müssen die Methoden, die überschrieben werden können, mit dem Schlüsselwort *virtual* deklariert werden. Die Möglichkeit Methoden zu Überschreiben, legt den Grundstein für die Polymorphie, die einen essenziellen Bestandteil der objektorientierten Programmierung darstellt. Dabei wird jede Klasse als Definition eines Datentyps angesehen. Ein Objekt einer Klasse besitzt als Datentypen seine Klasse und alle Basisklassen von denen seine ursprüngliche Klasse erbt [11]. Für das in Abbildung 2.2 dargestellte Beispiel bedeutet das, dass eine Instanz der Klasse *ConcreteClass* als Instanz vom Datentyp *AbstractClass* gespeichert werden kann. Beim Aufruf der überschriebenen Methode *virtualMethod()*, wird die Methode der Unterklasse anstelle der Methode der Basisklasse ausgeführt. Dadurch wird das Verwalten von Objektfamilien möglich, wobei jedes Objekt ein spezifisches Verhalten vorweisen kann. Die objektorientierte Programmierung erlaubt es zudem abstrakte Oberklassen zu erstellen. Eine abstrakte Klasse deklariert mindestens eine abstrakte Methode. Dabei handelt es sich um eine Methode die deklariert, aber nicht definiert ist. Das hat zur Folge, dass eine abstrakte Klasse nicht instanziierbar ist. Sie muss nach dem Prinzip der Vererbung abgeleitet werden, um in den Unterklassen Definitionen für die abstrakten Methoden bereitzustellen. In C++ werden abstrakte Methoden als rein virtuelle Methoden bezeichnet. Abstrakte Klassen und Methoden werden in einem UML-Klassendiagramm durch die kursive Schreibweise ihrer Namen hervorgehoben [9].

Assoziation

Die Assoziation ist eine Beziehung zwischen Klassen, die die Kommunikation der entsprechenden Instanzen verdeutlicht. Sie wird im UML-Klassendiagramm durch eine Verbindungslinie zwischen Klassen dargestellt. Es gibt verschiedene Ausprägungen der

Assoziation, die durch standardisierte Darstellungen repräsentiert werden könne. In der vorliegenden Arbeit wurde lediglich die einfach Assoziation verwendet. Die Instanzen der Klassen, die durch die Assoziation verbunden sind, können aufeinander zugreifen und Methodenaufrufe durchführen. Die Enden der Assoziation können mit Zahlenwerten beschriftet werden, die aussagen auf wie viele Instanzen der jeweiligen Klasse aufeinander zugreifen dürfen [10]. Das UML-Klassendiagramm in Abbildung 2.3 stellt eine Eltern-Kind-Beziehung dar. Demnach kann eine Instanz der Klasse *ExampleClass* auf beliebig viele weitere Instanzen der Klasse *ExampleClass* zugreifen. Die beliebig vielen Instanzen lassen sich wiederum auf eine bestimmte Instanz der Klasse zurückführen.

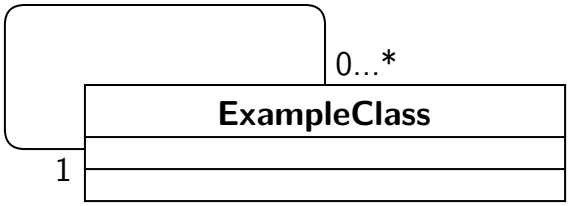

Abbildung 2.3
UML-Klassendiagramm zur Darstellung einer Eltern-Kind-Beziehung

Abhängigkeit

Die Abhängigkeit beschreibt eine Beziehung, in der eine Klasse von einer anderen Klasse abhängig ist. Werden Änderungen an einer Klasse durchgeführt, können diese Auswirkungen auf die von ihr abhängigen Klassen haben. Demnach sind Generalisierung und Assoziation ebenfalls Abhängigkeiten. Deren Semantik bezieht sich jedoch auf konkrete Verhaltensweisen der objektorientierten Programmierung, weshalb sie gesondert dargestellt werden. Durch Abhängigkeiten können alle Beziehungen zwischen Klassen visualisiert werden, für die es keine syntaktische Standardisierung von UML gibt. Sie werden durch eine gestrichelte Linie mit einer Pfeilspitze dargestellt, die auf die unabhängige Klasse verweist. Die Abhängigkeit kann durch eine Beschriftung des Pfeils genauer spezifiziert werden [9]. In Abbildung 2.4 bedeutet die Abhängigkeit, dass die Klasse *Producer* von der Klasse *Product* abhängig ist und dass sie Instanzen der Klasse *Product* erstellen kann.

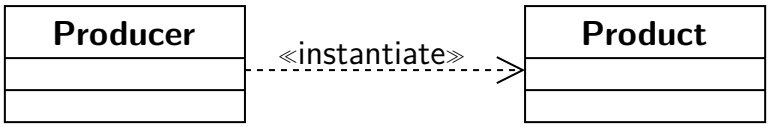

Abbildung 2.4
UML-Klassendiagramm zur Darstellung der Abhängigkeit zwischen Klassen

3 Aufgabenstellung

Ziel dieser Arbeit ist die Konzeptionierung und Implementierung eines Systems, welches das Erstellen von Prozessketten in GTlab mit Hilfe Python-Skripten ermöglicht. Im ersten Teil der Arbeit soll ein Verständnis für den vorliegenden Aufbau der GTlab Prozesssteuerung erarbeitet werden. Dazu sind der bislang fest vorgegebene hierarchische Aufbau und die verfügbaren Schnittstellen der Prozesskettenerstellung zu betrachten. Zu diesen Schnittstellen zählen die Prozesselemente, das zentrale Datenmodell und die zentrale Ressourcenverwaltung. Bei der Konzeptionierung des Systems, zur Erstellung von Prozessketten mit Hilfe von Python-Skripten, ist zu beachten, dass das System das Erstellen von Python-Objekten ermöglicht, die den in GTlab vorhandenen Prozesselementen entsprechen. Über diese Objekte sollen die Prozesselemente konfigurierbar und deren Ein- und Ausgabeparameter miteinander verschaltbar sein. Das System soll zudem den Zugriff auf das zentrale Triebwerksdatenmodell, sowie die zentrale Ressourcenverwaltung gewährleisten. In der darauffolgenden Implementierungsphase muss beachtet werden, dass das Konzept als eigenständige Funktionalität implementiert wird. Die bisherige Funktionsweise der Prozesssteuerung, über ein fest vorgegebenes hierarchisches Prinzip, soll erhalten bleiben, so dass beide Systeme parallel nutzbar sind. Um die nach diesem Prinzip erstellten Prozessketten in der entstehenden Python-Umgebung nutzen zu können, sollen diese automatisiert in ein entsprechendes Python-Skript umwandelbar sein. Abschließend soll das entwickelte Verfahren einem Anwendungstest unterzogen werden, wofür ein vorliegender Teilprozess der Triebwerksauslegung herangezogen wird. Dieser Test soll die Flexibilität der neu aufgebauten skriptbasierten Funktionalität, gegenüber dem bisherigen hierarchischen Aufbau demonstrieren.

4 Triebwerkvorentwurfsumgebung GTlab

Das Programmsystem GTlab ist eine interaktive Simulations- und Vorentwurfsumgebung für Flugtriebwerke und Gasturbinen. Es unterstützt die Entwicklung neuer Triebwerkskonzepte von den ersten thermodynamischen Berechnungen bis hin zu Ermittlung dreidimensionalen Geometriedaten und deren Visualisierung. Zudem kann GTlab dazu eingesetzt werden bestehende Triebwerke und ihr Betriebsverhalten zu bewerten. GTlab wurde in der Programmiersprache C++ geschrieben und basiert auf einer hochmodularen Architektur. Erweiterungen lassen sich über Module realisieren und in GTlab integrieren. Dabei stehen zahlreiche Schnittstellen zur Verfügung, über die beispielsweise neue Rechenverfahren oder Erweiterungen der grafischen Benutzeroberfläche implementiert werden können. Im Folgenden werden die drei Basismodule Performance, PreDesign und Sketchpad und ihre unterschiedlichen Aufgabenbereiche näher erläutert.

- Das Modul *Performance* berechnet das thermodynamischen Verhalten eines Triebwerks oder einer Gastrubine während des Betriebs. Über die grafische Benutzeroberfläche von GTlab lassen sich dazu verschiedene Komponenten eines Triebwerks miteinander verknüpfen. Das Modul nutzt komponentenspezifische Berechnungsmodelle, um das thermodynamische Gesamtverhalten des erstellten Triebwerks zu berechnen.

- Das *Sketchpad*-Modul ermöglicht die konzeptionelle Abschätzung erster Eigenschaften eines Triebwerks, wie beispielsweise Gewicht und Abmessungen. Anhand der in diesem Modul auf einfache und schnelle Weise berechneten Daten, kann der Einfluss des thermodynamischen Prozesses auf die Geometrie eines Triebwerks bewertet werden. Die entstandene Geometrie kann als Einstieg für einen

weiterführenden Detailentwurf dienen.

- Das Modul *PreDesign* bietet die Möglichkeit Berechnungsverfahren in GTlab zu integrieren, die zur Simulation der Triebwerkskomponenten dienen. Zudem lassen sich mit Hilfe dieses Moduls Ergebnisdaten vorangegangener Berechnungen auf verschiedene Weise visualisieren und auswerten.

In diesem Kapitel werden die Programmkomponenten von GTlab betrachtet, die für diese Arbeit von Relevanz sind. Zu diesen zählt das zentrale Datenmodell. Es werden dessen Datenumfang und dessen hierarchischer Aufbau, der unter Nutzung der Eltern-Kind-Beziehung der Qt-Bibliothek erstellt wurde, beschrieben. Zudem wird die zentrale Ressourcenverwaltung von GTlab erläutert, über die beispielsweise Materialdaten an alle GTlab-Benutzer verteilt werden können. Des Weiteren wird die bestehende Prozesssteuerung von GTlab mit ihren ausführbaren Komponenten betrachtet. Es wird eine Analyse des Aufbaus und des Ablaufs der Prozesssteuerung durchgeführt, um die daraus resultierenden Erkenntnisse für die Konzeptionierung der Python-basierten Prozesssteuerung zu nutzen. In diesem Kapitel werden UML-Klassendiagramme abgebildet. Es ist zu beachten, dass es sich bei den Diagrammen nicht um vollständige Abbilder der Programmkomponenten handelt. Auf Grund der Übersichtlichkeit beziehen sie sich lediglich auf die für dieses Kapitel relevanten Eigenheiten von GTlab.

4.1 Zentrales Datenmodell

GTlab nutzt ein zentrales Datenmodell, um die Daten eines Triebwerksvorentwurfs zu speichern. Jedes GTlab-Projekt, in dem der Vorentwurf eines Triebwerks durchgeführt wird, besitzt eine eigenständige Instanz dieses Datenmodells. Der Aufbau folgt dabei dem Prinzip eines dreidimensionalen Datenwürfels (siehe Abbildung 4.1) . Die erste Dimension bildet die einzelnen Komponenten eines Flugtriebwerks ab. Im Datenmodell können zu jeder dieser Komponenten spezifische Daten hinterlegt werden. Die Unterschiedlichen Disziplinen des Vorentwurfsprozess, wie beispielsweise Thermodynamik und

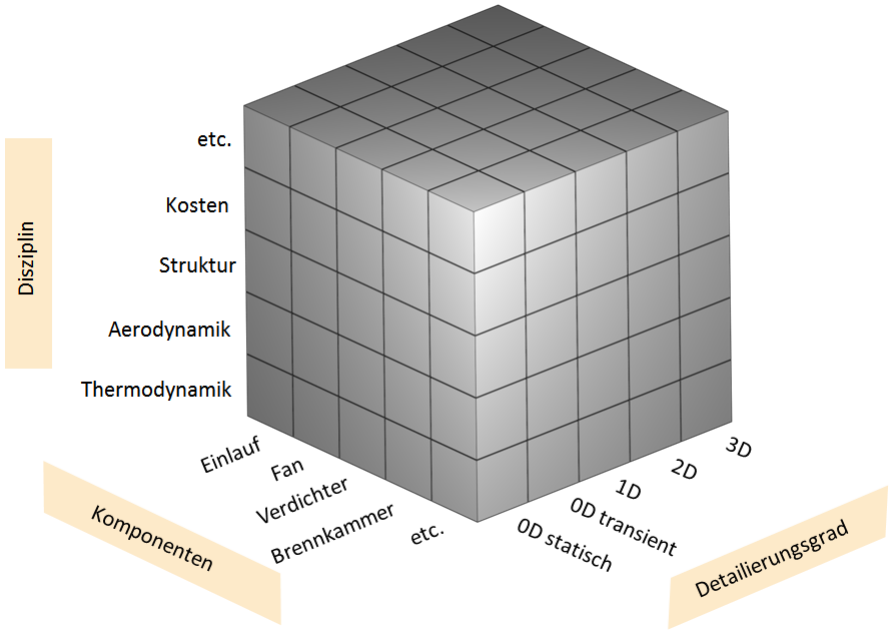

Abbildung 4.1
Abstrakte Darstellung des zentralen Datenmodells für den Triebwerksvorentwurf [12]

Aerodynamik, spannen die zweite Dimension auf. Die dritte Dimension bezieht sich auf den Grad der Genauigkeit der Daten. Da in diesem Datenmodell zu jeder der Komponenten eines Triebwerks, Daten unterschiedlicher Disziplinen und unterschiedlicher Genauigkeit hinterlegt sein können, lässt es sich auch als abstraktes und virtuelles Abbild des vorentworfenen Triebwerks betrachten. Dass es sich um ein *zentrales* Datenmodell handelt, ist daran zu erkennen, dass jegliche Module und Prozesselemente von GTlab innerhalb eines Projekts auf dasselbe Datenmodell zugreifen und ihre Daten über dieses transferieren [12]. Beim Ausführen von Berechnungen werden die benötigten Daten ausgelesen und die Ergebnisse im Datenmodell gespeichert. Diese Vorgehensweise stellt sicher, dass direkte Kommunikationswege zwischen Berechnungsverfahren und damit Abhängigkeiten verhindert werden. Zudem standardisiert es für alle GTlab-Module die Bezeichnungen und die Einheiten verschiedenster Parameter und den Zugang zu diesen.

Der Aufbau des zentralen Datenmodells folgt einem hierarchischem Prinzip, welches

dem allgemeinen Aufbau eines Triebwerks nachempfunden ist. Dazu existiert für jede Komponente des Triebwerks eine Datenmodellklasse. Instanzen dieser Klassen werden unter Nutzung der Eltern-Kind-Beziehung der Qt-Bibliothek in einer Baumstruktur hierarchisiert. Als Basisklasse der Datenmodellklassen dient die *GtObject*-Klasse (siehe Abbildung 4.2). Sie ist von der *QObject*-Klasse abgeleitet, wodurch alle Datenmodellklassen in der Eltern-Kind-Beziehung verwaltet werden können. So lassen sich beispielsweise beliebig viele Rotorschaufelreihen als Kindelement eines Verdichters definieren. Bei dem Klassendiagramm in Abbildung 4.2 handelt es sich um eine simplifizierte Darstellung der Datenmodellhierarchie, bei der zu Gunsten der Übersicht, nicht alle Klassenstrukturen und Abstraktionsebenen abgebildet sind.

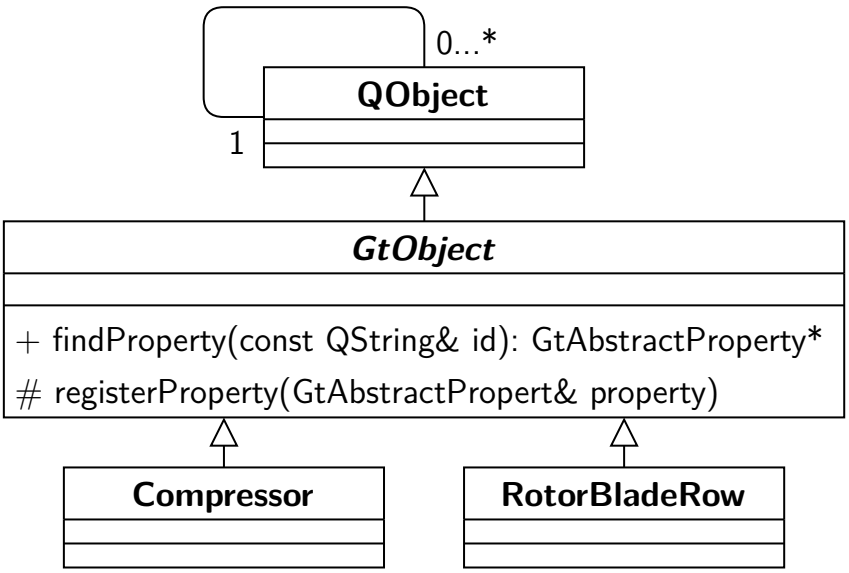

Abbildung 4.2
Eltern-Kind-Beziehung zur Hierarchisierung des zentralen Datenmodells

Das zentrale Datenmodell stellt ein Eigenschaftssystem bereit, über welches den Datenmodellklassen Eigenschaftswerte zugeordnet werden könne, die die physikalischen Eigenschaften der Triebwerkskomponente repräsentieren. Innerhalb der Konstruktoren der Datenmodellklassen können die entsprechenden Eigenschaften unter Verwendung der *registerProperty()*-Methode registriert werden. Dazu müssen die Eigenschaften

als Objekte vom abstrakten Datentyp *GtAbstractProperty* zu Verfügung stehen. Der Datentyp ist durch die abstrakte Klasse *GtAbstractProperty* in GTlab implementiert und kann durch Vererbung dazu verwendet werden, individuelle Eigenschaften zu definieren und sie auf bestimmte Datenmodellklassen zu spezialisieren (siehe Abbildung 4.3). Der Eigenschaftswert wird als Objekt der Klasse *QVariant* in den Instanzen vom Typ *GtAbstractProperty* hinterlegt. Die *QVariant*-Klasse wird von der Qt-Bibliothek bereitgestellt und dient als universeller Datentyp, der Werte aller gängigen Datentypen speichern kann [13]. Für die Standarddatentypen von C++ stellt GTlab bereits konkrete Implementierungen der *GtAbstractProperty*-Klasse zur Verfügung.

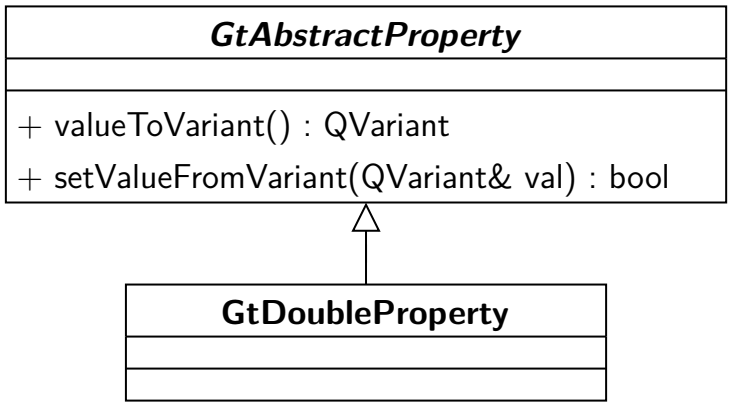

Abbildung 4.3
Abstrakte Klasse als Schnittstelle zur Implementierung von Eigenschaften in GTlab

4.2 Zentrale Ressourcenverwaltung

GTlab stellt eine Infrastruktur zur Verteilung zentral verwalteter Ressourcen bereit. Diese Ressourcen sind auf einem Server hinterlegt und können über die GTlab-Benutzeroberfläche heruntergeladen werden. So lassen sich beispielsweise Materialeigenschaftsdaten heranziehen und in die Berechnungen einfließen [14]. Der Benutzer hat dabei die Berechtigung zum lesenden, aber nicht zum schreibenden Zugriff. Dadurch wird die zentrale Verwaltung sichergestellt. Die Ressourcen unterliegen einer Versionierung, so dass ein Benutzer darauf hingewiesen wird, sobald neue beziehungsweise veränderte Daten bereitstehen.

4.3 Die Prozesssteuerung von GTlab

Um einen Triebwerksvorentwurf anfertigen zu können, müssen diverse Berechnungen durchgeführt werden, die sich auf die einzelnen Triebwerkskomponenten und deren Disziplinen beziehen. In GTlab ist die interne Prozessteuerung für die Durchführung solcher Berechnungen verantwortlich. Sie ermöglicht es verschiedenste Berechnungsverfahren zu verschalten und in Form von Prozessketten auszuführen. Dabei wird das zentrale Datenmodell als Datenquelle und als Speicherort für Ergebnisdaten herangezogen. Dieses Unterkapitel analysiert den vorliegenden Aufbau der Prozesssteuerung von GTlab. Dazu müssen zunächst die ausführbaren Komponenten von GTlab betrachtet werden. Deren Funktionsweise und Implementierung werden in diesem Unterkapitel beschrieben. Zudem werden der Aufbau und der Ablauf der Prozesssteuerung skizziert.

4.3.1 Ausführbare Komponenten in GTlab

Zu den ausführbaren Komponenten von GTlab zählen Tasks und Prozesselemente. Jede Prozesskette besteht aus mindestens einem Task, der als Wurzel zum Aufbau einer Prozesskette dient. Die Tasks beeinflussen den Ablauf der Berechnungen und dienen als Container für Prozesselemente und weitere Tasks. Die Prozesselemente beinhalten verschiedenste Algorithmen und Berechnungsverfahren, die für den Triebwerksvorentwurf benötigt werden. GTlab bietet die Möglichkeit Tasks zu erstellen und ihnen beliebige Prozesselemente oder weitere Tasks unterzuordnen. Wird eine Task gestartet, sorgt dieser dafür, dass die untergeordneten Prozesskomponenten ausgeführt werden. Ein Prozesselement ist nur als Unterelement eines Tasks und nicht eigenständig ausführbar. In Abbildung 4.4 ist ein Ausschnitt der grafischen Benutzeroberfläche von GTlab zusehen. Dieser zeigt die Darstellung einer Prozesskette in der GTlab-Benutzeroberfläche. Es ist zu erkennen, dass eine Task mit dem Namen *Single Operating Points* erstellt wurde. Diesem wurden zwei Prozesselemente zur Berechnung des thermodynamischen Kreisprozesses eines Triebwerks definiert. Auf eine Beschreibung der genauen Berechnung der dargestellten Prozesskette wird an dieser Stelle verzichtet, da die Prozesskomponenten im Allgemeinen betrachtet werden sollen. Die Abbildung 4.4 verdeutlicht die hierarchische

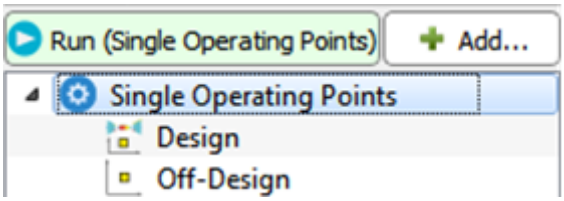

Abbildung 4.4
Darstellung einer Prozesskette in GTlab

Beziehung der Komponenten. Ein Task verwaltet die untergeordneten Prozesselemente. Die Hierarchie der Prozesskomponenten wird durch das UML-Klassendiagramm in Abbildung 4.5 deutlich. Es ist zu erkennen, dass die *GtProcessComponent*-Klasse von der *GtObject*-Klasse abgeleitet ist. Daher können Prozesskomponenten die in Abschnitt 4.1 beschriebene Schnittstelle *GtAbstractProperty* verwenden. Über diese können Eigenschaftswerte registriert werden, die von einem spezialisierten Datentypen abstammen. Bei den Eigenschaftswerten handelt es sich um die Eingabeparameter, die bei der Ausführung der entsprechenden Komponenten berücksichtigt werden sollen. GTlab stellt weitere Schnittstellen bereit, über die für jede Komponente individuelle Benutzeroberflächen implementiert werden können. Sie können dazu genutzt werden, Eingabewerte für die registrierten Eigenschaften grafisch abzufragen. Die *GtProcessComponent*-Klasse dient als Oberklasse für alle ausführbaren Komponenten der Prozessteuerung. Die Klasse *GtTask* repräsentiert die Tasks, die Klasse *GtCalculator* die Prozesselemente. Als Oberklasse beinhaltet die *GtProcessComponent*-Klasse alle gemeinsamen Eigenschaften der Prozesskomponenten. Sie beinhaltet beispielsweise Funktionen zum Setzen und Abfragen des Ausführungsstatus der Komponenten. Zudem deklariert sie die Methode *exec()*. Bei ihr handelt es sich um eine rein virtuelle Methode, die deklariert, jedoch nicht definiert ist. Aus Abbildung 4.5 geht hervor, dass Tasks und Prozesselemente von der *GtProcessComponent*-Klasse abgeleitet sind und somit eine Implementierung der *exec()*-Methode beinhalten müssen. In dieser Methode wird das Verhalten der Prozesskomponenten während der Ausführung bestimmt. Alle Tasks und Prozesselemente können unter dem abstrakten Datentyp *GtProcessComponent* verwaltet werden. Da innerhalb der *GtProcessComponent*-Klasse die Methode *exec()* deklariert ist, lässt sich diese auf alle *GtProcessComponent*-Instanzen anwenden. Wie in dem Klassendia-

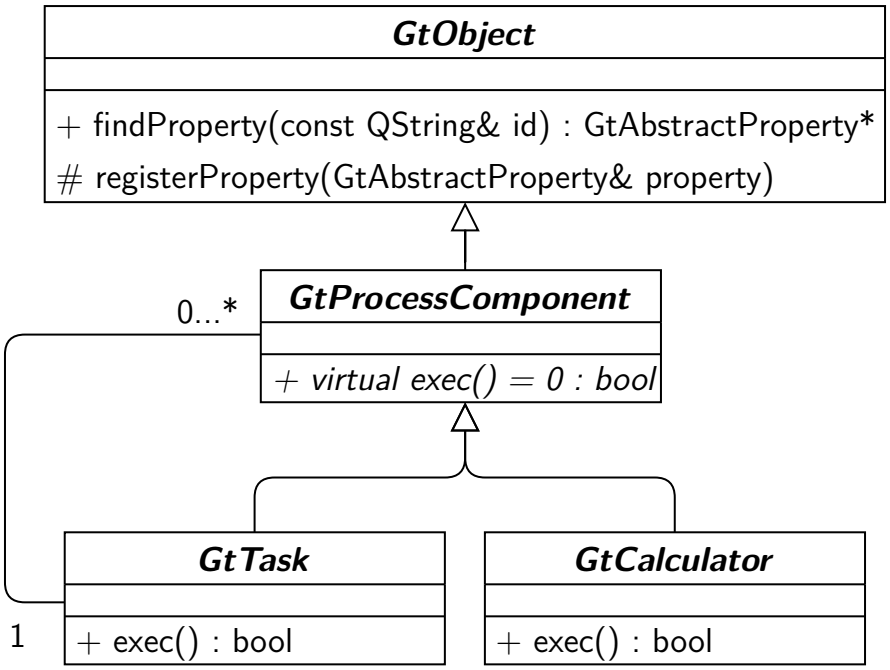

Abbildung 4.5
Hierarchische Beziehung von Prozesskomponenten

gramm zu erkennen, können Tasks mehrere Instanzen vom Typ *GtProcessComponent* über die Eltern-Kind-Beziehung verwalten. Bei der Ausführung eines Tasks werden die zugehörigen *exec()*-Methoden aller zugeordneten Prozesskomponenten hierarchisch aufgerufen. Im Weiteren wird die Implementierung der Tasks und der Prozesselemente näher betrachtet.

4.3.1.1 Tasks

Die Tasks stellen wichtige Komponenten zur Ausführung von Berechnungsverfahren und Algorithmen in GTlab dar. Sie entsprechen Containern, denen Prozesselemente und weitere Tasks untergeordnet werden können. Ein Task ist ein ausführbares Element und kann von der Prozesssteuerung gestartet werden. Daraufhin startet der Task, die untergeordneten Komponenten. Auf diese Weise lassen sich Prozesselemente beliebig hintereinander schalten und ausführen. Ein Standardtask führt dabei die untergeordneten Komponenten sequenziell aus. Bei der Programmierung eines Tasks kann dessen

Verhalten über die vordefinierten Schnittstellen frei implementiert werden. Somit besteht beispielsweise die Möglichkeit, Komponenten iterativ oder unter verschiedenen Voraussetzungen ausführen zu lassen. Die Prozesssteuerung sorgt vor der Ausführung eines Tasks dafür, dass der Zugriff auf die Daten des zentralen Datenmodells besteht und Daten in diese zurückführt werden können.

Im GTlab-Programmcode sind Tasks durch die *GtTask*-Klasse realisiert (siehe Abbildung 4.6). Die *GtTask*-Klasse beinhaltet eine Implementierung der Methode *exec()*, die zum Starten des Tasks aufgerufen wird. Zudem werden virtuelle Methoden bereitgestellt, die innerhalb einer Unterklasse neu implementiert werden können. Die *setUp()*-Methode wird vor jeder Ausführung aufgerufen und kann dazu verwendetet werden, initiale Eigenschaften der zu definieren. Der Rückgabewert der Methode bestimmt, ob der Task ausgeführt wird. Das Ausführungsverwalten des Tasks ist in der *runIteration()*-Methode definiert. Die standardmäßige Implementierung nutzt die *runChildElements()*-Methode, um die untergeordneten Prozesskomponenten, durch den Aufruf derer *exec()*-Methoden, sequenziell auszuführen. In der *runChildElements*-Methode wird nach der Ausführung der Prozesskomponenten die Methode *evaluate()* aufgerufen. In dieser können die Ergebnisdaten evaluiert und der Status des Tasks zurückgegeben werden. Durch das Erstellen einer Unterklasse der *GtTask*-Klasse und das Überschreiben der virtuellen Methoden,

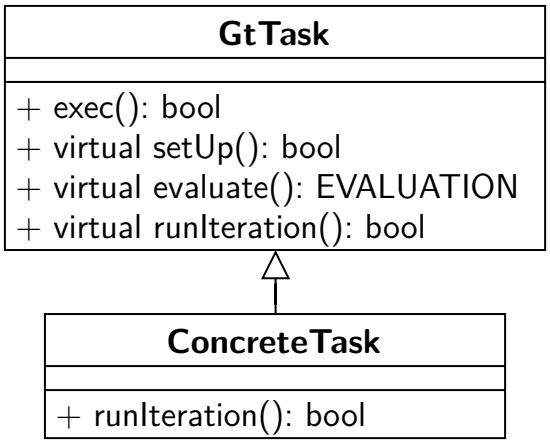

Abbildung 4.6
Schnittstelle zur Implementierung von Tasks

kann das Ausführungsverhalten eines Tasks frei definiert werden.

4.3.1.2 Prozesselemente

Im multidisziplinären Triebwerksvorentwurf wird eine große Anzahl verschiedenster Auslegungsverfahren eingesetzt. Diese Verfahren können auf das Gesamtverhalten des Triebwerks oder einzelne Triebwerkskomponenten und deren Disziplinen spezialisiert sein. In GTlab werden Berechnungsverfahren in Form von Prozesselementen bereitgestellt. Als Prozesselemente werden voneinander unabhängige Programmkomponenten bezeichnet, in denen verschiedenste Algorithmen implementiert sein können. Ein Prozesselement kann beispielsweise einen Algorithmus beinhalten, der dazu dient Geometrieberechnungen bezogen auf eine Triebwerkskomponente durchzuführen. Dabei müssen Prozesselemente nicht zwangsläufig Berechnungsverfahren bereitstellen. Ihre Algorithmen sind für den Programmierer frei definierbar. So besteht ein weiterer Anwendungsfall für Prozesselemente darin, Daten aus dem zentralen Datenmodell in Dateien von unterschiedlichem Format zu exportieren. Die Prozesselemente sind nicht eigenständig ausführbar, sondern lassen sich nur als Unterelement eines Tasks ausführen. Um einem Task Prozesselemente unterzuordnen, erlaubt die GTlab Benutzeroberfläche alle verfügbaren Prozesselemente einzusehen und auszuwählen. Je nach Implementierung eines Prozesselements, ist es möglich benutzerdefinierte Eingabeparameter festzulegen. Die zum Teil zwingend notwendigen und zum Teil optionalen Eingabeparameter werden bei der Ausführung des Algorithmus des Prozesselements berücksichtigt. Zudem ermöglicht die Prozesssteuerung, während der Ausführung eines Prozesselements Daten aus dem zentralen Datenmodell heranzuziehen und Ergebnisdaten in dieses zurückzuführen. Auf diese Weise können Prozesselemente Vorentwurfsdaten verwenden, die durch vorangegangene Prozesselemente entstanden sind. Sie können somit über das zentrale Datenmodell interagieren, ohne eine direkte Kommunikation miteinander aufbauen zu müssen.

GTlab bietet definierte Schnittstellen zur Erstellung und Integration neuer Prozesselemente an. Diese erlauben es neue Algorithmen in GTlab zu integrieren, die Berechnungen oder sonstige Aufgaben innerhalb einer Prozesskette übernehmen. In Abbildung 4.7 ist die

Schnittstelle zur Erstellung neuer Prozesselemente in Form eines UML-Klassendiagramms dargestellt. Die *GtCalculator*-Klasse dient als Oberklasse für alle Prozesselemente und beinhaltet alle Funktionalitäten, die ein Prozesselement benötigt, um als Teil einer Prozesskette ausführbar zu sein. Die *GtCalculator*-Klasse ist als abstrakte Klasse implementiert und beinhaltet daher die rein virtuelle Funktion *run()*. Um ein konkretes Prozesselement zu erstellen, muss eine Klasse erstellt werden, die von der *GtCalculator*-Klasse erbt und somit dessen Funktionalitäten übernimmt. Damit ist sichergestellt, dass alle Prozesselemente dieselben Grundfunktionalitäten vorweisen. Sie unterscheiden sich in der Art des Algorithmus, der nach dem Starten des Prozesselements ausgeführt wird. Um den elementspezifischen Algorithmus zu definieren, muss innerhalb eines konkreten Prozesselements die deklarierte *run()*-Methode implementiert werden. Der entsprechende Algorithmus in dieser Methode ist für den Programmierer frei definierbar. Dabei kann auf Datenmodellobjekte zugegriffen werden, um dessen Werte auszulesen oder zu verändern. Die *run()*-Methode wird automatisiert innerhalb der *exec()*-Methode der Oberklasse *GtCalculator* aufgerufen. Auf diese Weise ist garantiert, dass bei der Ausführung eines Prozesselements innerhalb einer Prozesskette, der für das Prozesselement spezifisch definierte Algorithmus aufgerufen wird.

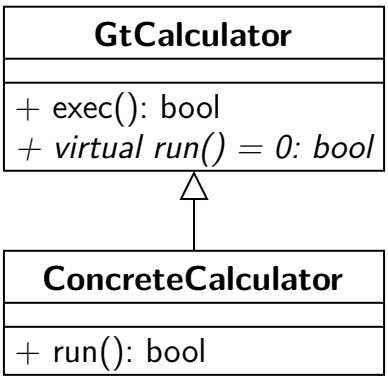

Abbildung 4.7
Schnittstelle zur Implementierung von Prozesselementen

4.3.2 Aufbau und Ablauf der Prozesssteuerung

Die Prozesssteuerung von GTlab ist dafür verantwortlich, dass eine Prozesskette korrekt ausgeführt wird und dabei der Zugriff auf das zentrale Datenmodell möglich ist. Ein Prozesskette wird durch einen Task und die untergeordneten Komponenten definiert. Je nach Anzahl von Iterationen und der Komplexität der Unterkomponenten, kann die benötigte Rechenleistung und die Zeit, die zum Ausführen eines Prozesses benötigt wird, stark variieren. Daher ist es Aufgabe der Prozesssteuerung, die GTlab-Benutzeroberfläche während der Ausführung eines Prozesses durch Parallelisierung weiterhin bedienbar zu halten.

Das UML-Klassendiagramm in Abbildung 4.8 stellt den Aufbau der Prozessteuerung von GTlab dar. Die Prozessteuerung besteht aus drei Klassen, die miteinander agieren, um Prozesse auszuführen. Als zentrale Verwaltung der Prozesssteuerung kann die *GtProcessExecutor*-Klasse betrachtet werden. Deren *runTask()*-Methode erwartet als Übergabewert die Task-Instanz, die ausgeführt werden soll. Sie instanziiert die beiden weiteren Klassen und nutzt deren Funktionalität zur Ausführung der Task-Instanz. Die *GtTaskRunner*-Klasse sorgt dafür, dass dem Prozess während der Ausführung eine Kopie des zentralen Datenmodells zur Verfügung steht. Zudem fordert sie einen Thread an

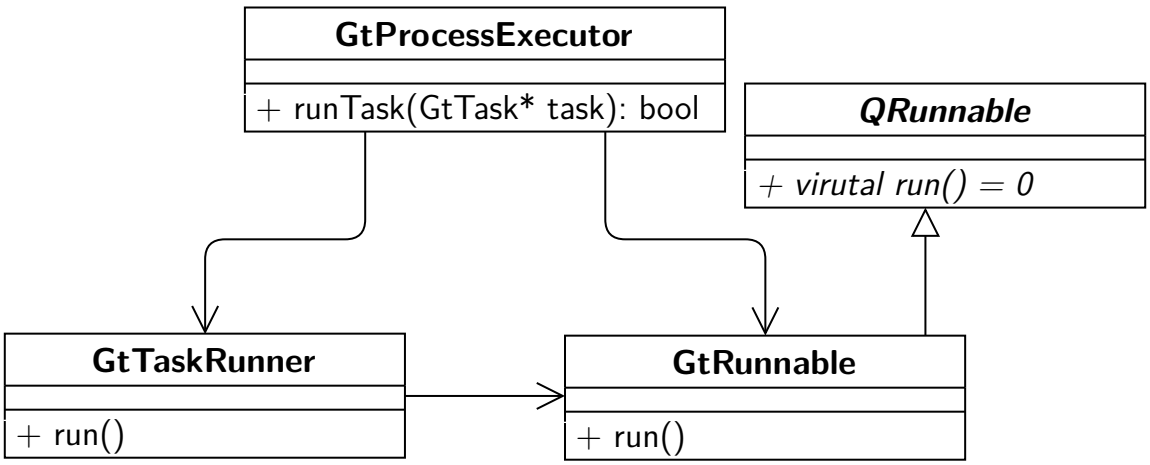

Abbildung 4.8
Aufbau der Prozesssteuerung von GTlab

und übergibt diesem den auszuführenden Prozess. Der Prozess wird durch die Klasse *GtRunnable* repräsentiert. Sie stellt die Komponente dar, die an einen Thread übergeben und innerhalb dieses ausgeführt werden kann.

Der Ablauf der Prozesssteuerung ist in Abbildung 4.9 vereinfacht dargestellt. Darin werden die einzelnen Schritte verdeutlicht, die bei der Ausführung von Prozessen abgearbeitet werden. Es ist auch zu entnehmen, an welcher Stelle diese im Programmcode implementiert sind. Nach dem Start eines Prozesses, wird die Instanz der *GtProcessExecutor*-Klasse angefordert. Diese Klasse wurde unter Verwendung des Singleton-Entwurfsmusters erstellt, welches sicherstellt, dass nur eine Instanz der *GtProcessExecutor*-Klasse existieren kann [15]. An diese Instanz wird der selektierte Task durch den Aufruf der *runTask()*-Methode übergeben. Es folgt die Instanziierung der *GtRunnable*-Klasse und der *GtTaskRunner*-Klasse. Die *GtTaskRunner*-Instanz wird mit der Task-Instanz initialisiert. Zudem wird die *GtTaskRunner*-Instanz das Datenmodell und die erzeugte *GtRunnable*-Instanz übergeben. Die *GtTaskRunner*-Instanz, erstellt eine Kopie des Datenmodells und der Task-Instanz und leitet diese an die *GtRunnable*-Instanz weiter. Während der Ausführung des Prozesses können Daten aus der Datenmodellkopie ausgelesen und modifiziert werden. Es wird deutlich, dass die *GtRunnable*-Instanz, die den auszuführenden Prozess darstellt, lediglich mit einer Kopie des zentralen Datenmodells in Kontakt kommt. Sollte die Ausführung eines Prozesses fehlschlagen, ist die Konsistenz des Datenmodells nicht gefährdet. Um die Ausführung des Prozesses in einem gesonderten Thread zu starten, wird das Konzept *QRunnable* und *QThreadPool* der Qt-Bibliothek genutzt. *QRunnable* stellt eine abstrakte Klasse dar, die die rein virtuelle Methode *run()* definiert. Die *GtRunnable*-Klasse implementiert die *run()*-Methode so, dass in ihr die *exec()*-Methode des kopierten Tasks aufgerufen wird. Die *GtTaskRunner*-Instanz übergibt die *GtRunnable*-Instanz an die *QThreadPool*-Instanz. Daraufhin wird die *run()*-Methode und damit auch der übergebene Task in einem separierten Thread gestartet. Wurde der Prozess abgeschlossen, überführt die *GtProcessExecutor*-Instanz die Änderungen, die an der Kopie des Datenmodells vorgenommen wurden, in das zentrale Datenmodell. Damit endet die Ausführung eines Prozesses.

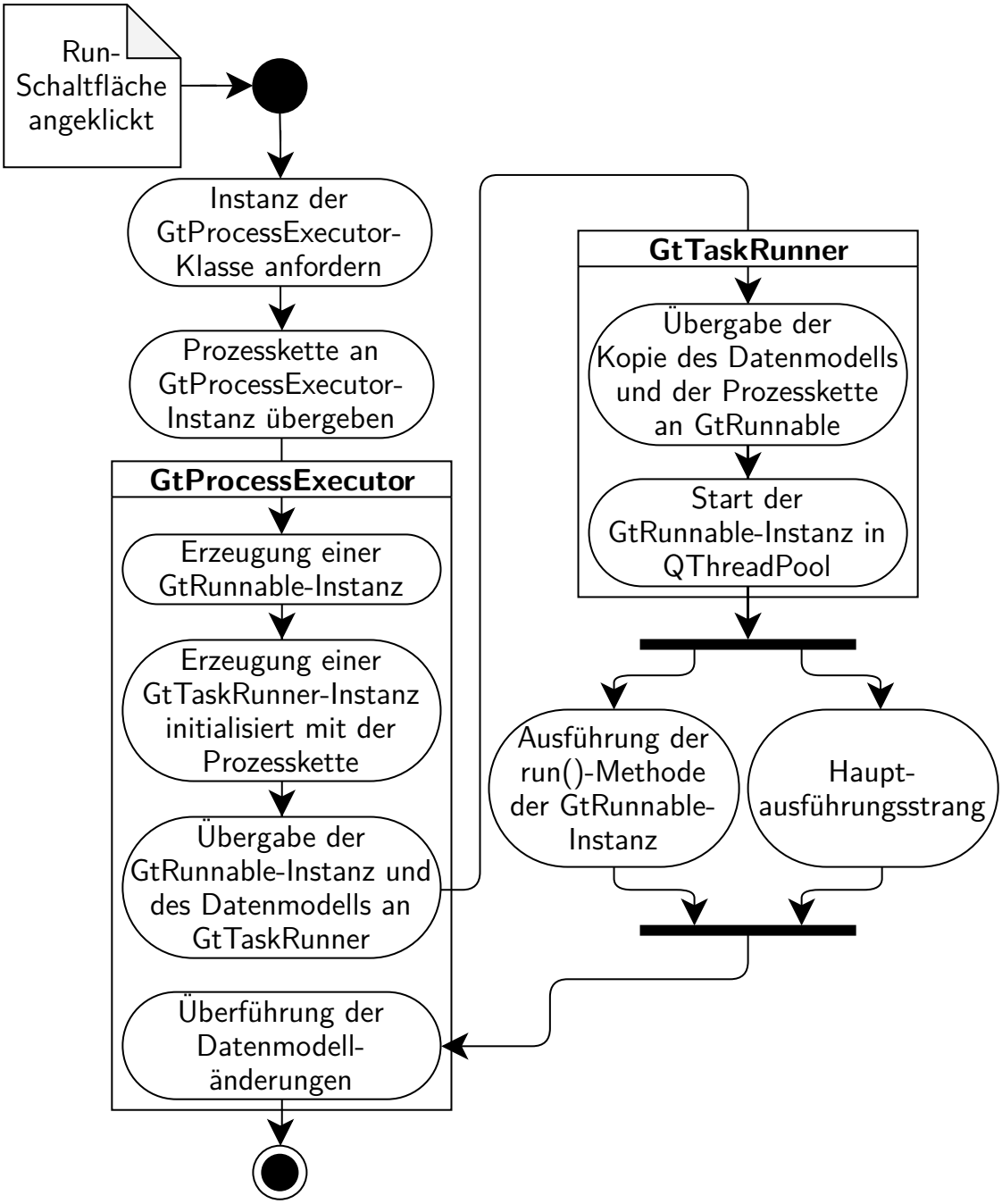

Abbildung 4.9
Vereinfachter Arbeitsweise der Prozessteuerung von GTlab

4.4 Die Python-Schnittstelle in GTlab

GTlab bietet eine Schnittstelle an, die es ermöglicht auf einen Python-Interpreter zuzugreifen und diesen zu nutzen. Um den Zugriff auf den Python-Interpreter zu ermöglichen, ist die Klassenbibliothek PythonQt in GTlab integriert. Da GTlab aus verschiedenen Klassenbibliotheken besteht, die beim Kompilieren zu einem lauffähigen Programm zusammengeführt werden, ist die Python-Schnittstelle so definiert, dass möglichst wenige Abhängigkeiten zur PythonQt-Bibliothek aufgebaut werden (siehe Abbildung 4.10) . Die Implementierung des Fassade-Entwurfsmusters ermöglicht eine effiziente Integration von PythonQt in GTlab. Das Fassede-Entwurfsmuster wird durch eine Klasse realisiert, die eine vereinfachte Schnittstelle zur Nutzung eines zusammenhängenden Klassensystems darstellt [15]. In GTlab wird die Klasse *GtPythonContextManager* als zentrale Schnittstelle zur PythonQt-Bibliothek genutzt. Diese Klasse befindet sich innerhalb der Kern-Klassenbibliothek. Die Kern-Klassenbibliothek beinhaltet alle grundlegenden Funktionalitäten von GTlab. Die meisten Klassenbibliotheken aus denen GTlab besteht, stehen in Abhängigkeit zur Kern-Klassenbibliothek. Damit sind deren Funktionalitäten in den meisten Teilen des Programmcodes verfügbar. Die PythonQt-Funktionalitäten werden an drei Stellen verwendet, die sich in unterschiedlichen Klassenbibliotheken befinden. Zum einen wird sie innerhalb einer GTlab-Konsolenanwendung genutzt, die beim Start als Übergabeargument einen Dateipfad zu einer Python-Skript-Datei erwartet.

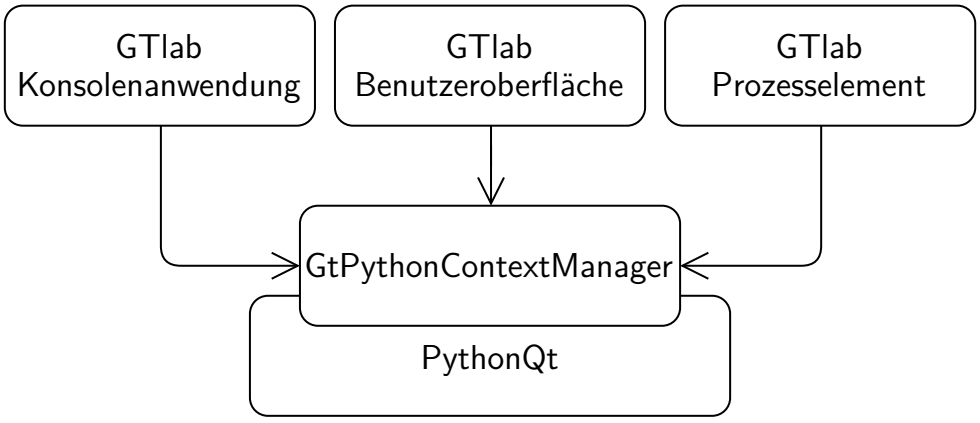

Abbildung 4.10
Veranschaulichung Schnittstelle zur Nutzung von PythonQt in GTlab

Diese Konsolenanwendung führt GTlab unter Berücksichtigung des angegebenen Skripts aus. So kann die Ausführung von GTlab über das Python-Skript gesteuert werden. In der GTlab-Benutzeroberfläche befindet sich eine Python-Konsole, die es ermöglicht GTlab zur Laufzeit über Python-Befehle zu steuern und Datenmodellanpassungen durchzuführen. Zudem wird die PythonQt-Funktionalität von einem Prozesselement genutzt. Das Prozesselement erlaubt es ein Berechnungsverfahren in Form eines Python-Skripts zu definieren. Dabei kann auf das zentrale Datenmodell zugegriffen werden, um Datenmodellwerte auszulesen und zu modifizieren. Beim Ausführen des Prozesselements innerhalb einer Prozesskette, wird das entsprechende Python-Skript an einen Python-Interpreter übergeben und ausgeführt. So ermöglicht das Prozesselement zur Laufzeit von GTlab benutzerdefinierte Berechnungsverfahren in Prozessketten zu integrieren. Insbesondere die prototypische Entwicklung neuer Prozesselemente wird dadurch vereinfacht. Die Python-Schnittstelle verwaltet zum Ausführen von Python-Skripten mehrere Python-Kontexte. Sie stellen Umgebungen dar, in denen Python-Skripte unabhängig voneinander ausgeführt werden können. Die entsprechende Umgebung speichert die im Skript definierten Variablen, Methoden und Klassen zur späteren Wiederverwendung. Sie lassen sich somit unabhängig voneinander mit vordefinierten Programmkonstrukten ausstatten und auf Anwendungsfälle spezialisieren. Die Schnittstelle ist so implementiert, dass sie in wenigen Schritten um einen Python-Kontext erweitert werden kann.

5 Integration einer skriptbasierten Prozesssteuerung in GTlab

In GTlab lassen sich Prozesskomponenten in Form von Prozessketten beliebig verschalten und ausführen. Die Ausführung erfolgt dabei in einer definierten hierarchischen Reihenfolge. Im Rahmen der vorliegenden Arbeit sollte die hierarchische Ausführung aufgebrochen und das Definieren von flexiblen Prozessketten in GTlab möglich werden. Dazu sollen Prozessketten zukünftig über Python-Skripte, unter Nutzung aller gängigen Python-Befehle und Anweisungen, aufgebaut werden können. Der in GTlab bereits integrierte Python-Interpreter ermöglicht dabei die Ausführung von Python-Code über eine fest definierte Schnittstelle. In diesem Kapitel wird beschrieben, wie die Python-Schnittstelle aus GTlab entfernt und in ein neu erstelltes GTlab-Modul überführt wurde. Auf diese Weise wurde die Abhängigkeit von GTlab zu Python aufgehoben. Die Python-Funktionalitäten können bei bedarf über das Python-Modul in GTlab integriert werden. Es werden die benötigten Erweiterungen der Python-Schnittstelle erläutert, die es ermöglichen Prozesselemente über Python-Skripte zu instanziieren und auszuführen. Die skriptbasierte Prozesssteuerung wurde über einen Task in GTlab realisiert. Deren Implementierung, sowie der Aufbau einer grafischen Benutzeroberfläche zum Erstellen und Ausführen von Prozesselementen werden ebenfalls in diesem Kapitel beschrieben.

5.1 Auslagerung der Python-Funktionalität in ein GTlab-Modul

Vor der Durchführung der vorliegenden Arbeit, stellte GTlab eine Python-Schnittstelle bereit, die als Teil der Kern-Klassenbibliothek implementiert war (siehe Abschnitt 4.4). Die Python-Schnittstelle stand in allen Programmteilen von GTlab zur Verfügung und konnte zur Ausführung von Python-Skripten verwendet werden. Zur Implementierung

der Python-Schnittstelle wurde Python fest in die Kern-Klassenbibliothek von GTlab verankert. Das hatte zur Folge, dass zur Ausführung von GTlab ein Python-Interpreter auf dem entsprechenden Rechner vorhanden sein musste. Damit sich GTlab unabhängig von Python ausführen lässt, wurde im Rahmen der vorliegenden Arbeit die bestehende Python-Schnittstelle aus der Kern-Klassenbibliothek in ein eigenes GTlab-Modul überführt. Ein GTlab-Modul stellt eine Klassenbibliothek dar, die als Plug-in dynamisch in GTlab integriert werden kann. Plug-ins erweitern Programme, ohne ein erneutes Kompilieren notwendig zu machen. Dazu muss das Programm Schnittstellen definieren, über die die Plug-ins eingeladen werden können [15]. In GTlab sind zahlreiche Schnittstellen zur Erweiterung von Berechnungsverfahren, Visualisierungsmethoden und Benutzeroberflächenkomponenten definiert, die innerhalb eines GTlab-Moduls implementiert werden können.

Um GTlab unabhängig von Python zu machen, musste die Abhängigkeit zur PythonQt Bibliothek aufgelöst und die Python-Schnittstelle aus der Kern-Bibliothek entfernt werden. Für die Python-Funktionalitäten wurde ein neues GTlab-Modul erstellt, welches im Weitern als Python-Modul bezeichnet wird. Das Python-Modul wurde mit einer Abhängigkeit zu PythonQt ausgestattet, um dessen Funktionalitäten innerhalb des Moduls nutzen zu können. Der Zugriff auf PythonQt wird im Python-Modul durch eine Python-Schnittstelle zentral verwaltet. Dazu wurde das Fassade-Entwurfsmuster angewandt, welches in Abschnitt 4.4 beschrieben ist. Als Schnittstelle zu PythonQt wurde die Klasse *GtpyContextManager* erstellt. Sie übernimmt die Aufgabe der *GtPythonContextManager*-Klasse, die vor der Erstellung des Python-Moduls die Python-Schnittstelle in GTlab darstellte. Alle Komponenten von GTlab, die Zugriff auf PythonQt benötigen, wurden in das Python-Modul überführt. Bei den Komponenten handelt es sich um die Python-Konsole, die in die GTlab-Benutzeroberfläche integriert ist und das Prozesselement, welches die Definition von Berechnungsverfahren in Form von Python-Skripten ermöglicht. Über entsprechende Modul-Schnittstellen wurden die überführten Komponenten für GTlab verfügbar gemacht. Zukünftig dient das Python-Modul als zentrale Sammlung der Python-Komponenten und bietet die Möglichkeit weitere Python-Funktionalitäten in GTlab zu integrieren. Alle Python-Komponenten, die außerhalb des

Python-Moduls, verteilt im GTlab Programmcode implementiert waren, konnten nach der Erstellung des Python-Moduls entfernt werden.

(Abb.)

5.2 Python-Kontext für die skriptbasierte Prozesssteuerung

Die Python-Schnittstelle von GTlab stellt verschiedene Kontexte zur Ausführung von Python-Code bereit. Sie stellen Umgebungen dar, in denen Python-Code unabhängig voneinander ausgeführt werden kann. Die Kontexte können über Python-Code mit Variablen, Methoden und Klassen vordefiniert werden. Der Python-Kontext, in dem die Skripte zur Steuerung von GTlab-Prozessen ausgeführt werden sollen, musste so vordefiniert sein, dass er das Erstellen und Ausführen von Prozesselementen in Form von Python-Objekten ermöglicht. Im Rahmen der vorliegenden Arbeit, wurde die Python-Schnittstelle des Python-Moduls um einen Python-Kontext erweitert, der die zuvor beschriebene Anforderung erfüllt. Im Weiteren wird beschrieben, wie der Python-Kontext vordefiniert wurde und welche Erweiterungen dazu innerhalb des Python-Moduls notwendig waren.

Um das Erstellen von Prozesselementen in einem Python-Kontext zu ermöglichen, muss die Art und die Anzahl der Prozesselemente bekannt sein. Die Anzahl der Prozesselemente hängt von den Modulen ab, die nach dem Programmstart von GTlab eingelesen werden. Das hat zur Folge, dass erst zur Laufzeit festgestellt werden kann, welche Prozesselemente zur Verfügung stehen. Die verfügbaren Prozesselemente werden beim Einlesen der Module in einer zentralen Sammlung registriert. Innerhalb der Sammlung befindet sich für jedes registrierte Prozesselement eine Instanz der *QMetaObject*-Klasse. Mit Hilfe der *QMetaObject*-Instanz, lassen sich neue Instanzen eines Prozesselemente erzeugen. Für den Python-Kontext wurde eine Klasse erstellt, die die beschriebene Sammlung nutzt, um neue Instanzen der Prozesselemente zu erzeugen. Dabei wurde eine Variante des Factory-Entwurfsmusters eingesetzt (siehe Abbildung 5.1). Das

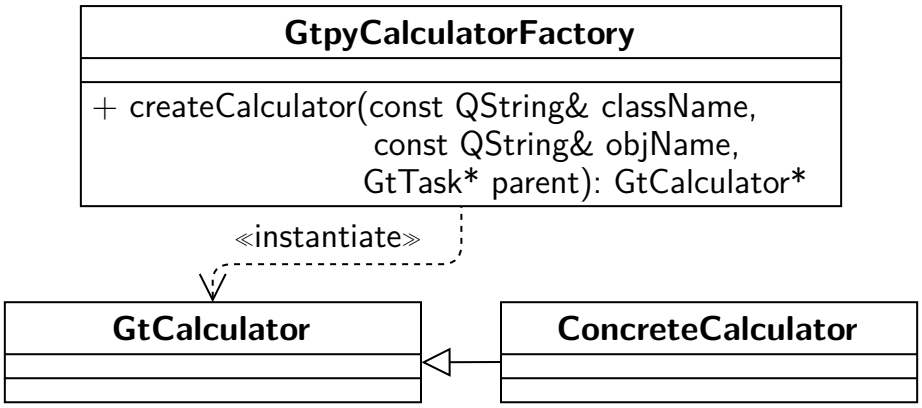

Abbildung 5.1
Implementierung der Factory-Klasse zur Erzeugung von Prozesselementen

Entwurfsmuster besteht aus einer Schnittstelle zur Erzeugung von Objekten. Die Objekte können von unterschiedlichen Klassen abstammen, die jedoch eine gemeinsame Oberklasse besitzen müssen. Durch den Einsatz des Factory-Entwurfsmusters kann im Programmcode darauf verzichtet werden, den konkreten Konstruktor einer Klasse aufzurufen. Die zu erstellende Klasse muss dazu in der Factory-Klasse registriert sein. Auf diese Weise lässt sich zur Laufzeit entscheiden, welche Klasse instanziiert werden soll [15]. Das erhöht die Flexibilität der Anwendung und vereinfacht die Einbindung neuer Klassen. Die erstellte Factory-Klasse wurde mit einer Methode ausgestattet, die das Instanziieren der Prozesselemente ermöglicht. Diese erwartet als Parameterwert den Klassennamen der zu erzeugenden Klasse. Anhand des Klassennamens kann die zugehörige *QMetaObject*-Instanz aus der zuvor beschriebenen Sammlung der Prozesselemente gefiltert werden. Die *QMetaObject*-Instanz wird zur Instanziierung des entsprechende Prozesselement verwendet. Zudem kann der Methode ein Objektname als Parameterwert übergeben werden. Der Objektname wird dem Prozesselement nach seiner Erzeugung zugeordnet. Jede Instanz von Typ *QObject* besitzt solch einen Objektnamen, der im Python-Kontext zur Identifikation der Instanz dient. Als dritter Parameter erwartet die Methode eine Elterninstanz, der das erstellte Prozesselement untergeordnet werden soll. Da ein Prozesselement nur als Kindelement einer Task-Instanz ausführbar ist, muss die übergebene Elterninstanz von der Klasse *GtTask* abstammen. Der neu erstellte Python-Kontext wurde mit der Instanz der Factory-Klasse ausgestattet. Dadurch wurde

das Instanziieren von Prozesselemente über Python-Code möglich.

Durch die Integration der Factory-Instanz in den Python-Kontext wurde die Anforderung, das Erstellen von Prozesselementen innerhalb von Python-Skripten zu ermöglichen, erfüllt. Daraufhin wurde die Nutzung der Factory-Instanz im Bezug auf die Benutzerfreundlichkeit betrachtet. In Abbildung 5.2 ist der Python-Programmcode dargestellt,

```
1 calc = CalculatorFactory.createCalculator('GtExampleCalculator', 'calcName',
      __task)
```

Abbildung 5.2
Python-Programmcode zur Erstellung eines Prozesselements im Python-Kontext

der zur Erstellung eines Prozesselements unter Nutzung der Factory-Instanz implementiert werden muss. Ein Programmierer muss dabei Kenntnisse über das Verhalten der Factory-Instanz besitzen. Zudem muss der Klassennamen des zu erzeugenden Prozesselements bekannt sein und in Form einer Zeichenkette an die Factory-Methode übergeben werden. Um die Benutzerfreundlichkeit zu erhöhen, wurde der Python-Kontext mit vordefinierten Methoden zur Erstellung der Prozesselemente ausgestattet (siehe Abbildung 5.3). Diese Methoden werden über den Klassennamen des entsprechenden Prozesselements aufgerufen. Sie sind so definiert, dass sie die Factory-Instanz Nutzen, um ein Prozesselement zu erzeugen und zurückzugeben . Bei der Entwicklung dieses Konzepts wurde darauf geachtet, dass GTlab je nach Konfiguration unterschiedliche Prozesselemente beinhalten kann. Die Definition der Methoden zum Erzeugen von

```
1 def GtExampleCalculator(name = 'ExampleCalculator'):
2
3     calc = CalculatorFactory.createCalculator('GtExampleCalculator', name,
          __task)
4
5     return CalcWrapper(calc)
```

Abbildung 5.3
Implementierung einer exemplarischen Methode zur Instanziierung eines Prozesselements

Prozesselementen wird daher nach jedem Start von GTlab dynamisch durchgeführt. Dazu wird die erwähnten Sammlung von verfügbaren Prozesselementen genutzt. Auf diese Weise ist sichergestellt, dass zukünftig entwickelte Prozesselemente automatisiert innerhalb des erstellten Python-Kontexts zur Verfügung stehen.

5.2.1 Prozesselemente im Python-Kontext

Zum Konfigurieren und Auslesen der Eingabeparameter, muss auf die individuell registrierten Eigenschaften der Prozesselemente zugegriffen werden. Die Schnittstelle für den Zugriff ist über die *setPropertyValue()* und die *propertyValue*-Methode definiert (siehe Abbildung 5.4). Diese setzten voraus, dass die identifizierenden Kürzel der entsprechenden Eigenschaften bekannt sind. Das Kürzel bestimmt, welcher Eigenschaftswert gesetzt oder ausgelesen werden soll. Durch die Vielzahl der Prozesselemente und die Individualität der Eigenschaften, ist eine Konfiguration eines Prozesselements über die genannten Methoden nur mit speziellen Kenntnissen oder einer Dokumentation der Prozesselemente möglich. Um Anwendern das Konfigurieren von Prozesselementen über Python-Skripte zu erleichtern, wurde eine umhüllende Klasse namens *CalcWrapper* geschaffen. Alle Prozesselemente im Python-Kontext sind von einer Instanz der umhüllenden Klasse umgeben (siehe Abbildung 5.3). Die Instanz delegiert Funktionsaufrufe an die entsprechende Prozesselement-Instanz. Solche Klassen werden als *Wrapper* bezeichnet. Sie verwalten den Zugriff auf die umhüllte Instanz und erweitern gegebenenfalls dessen Funktionalität [7]. In Abbildung 5.5 ist ein Teil der Klassendefinition der *CalcWrapper*-Klasse dargestellt. Es ist zu erkennen, dass innerhalb der *CalcWrapper*-Klasse die Methode *__init__()*

```
1 calc.setPropertyValue('ExampleProperty', 42)
2
3 prop = calc.propertyValue('ExampleProperty')
4
5 print (prop) #output: 42
```

Abbildung 5.4
Python-Programmcode zum Setzen und Auslesen von Eigenschaftswerten eines Prozesselements

überschrieben wurde. Diese Methode wird von Python automatisiert aufgerufen, wenn eine neue Instanz der entsprechenden Klasse erzeugt wird und dient somit als Konstruktor der Klasse. Die *__init__()*-Methode kann dazu verwendet werden, die erzeugte Instanz mit verschiedenen Attributen zu initialisieren. Dabei stellt der Parameter *self* die neu erstellte Instanz dar. In der *CalcWrapper*-Klasse wurde die *__init__()*-Methode so implementiert, dass das übergebene Prozesselement in der *CalcWrapper*-Instanz gespeichert wird. Für jede Eigenschaft des übergebenen Prozesselements, werden zwei Methoden definiert und der *CalcWrapper*-Instanz zugeordnet. Diese Methoden nutzen die *setPropertyValue()*- und *propertyValue()*-Methode eines Prozesselements und ermöglichen das Setzen und Auslesen der entsprechenden Eigenschaftswerte. Der Programmcode in Abbildung 5.6 verdeutlicht den daraus resultierenden Vorteil. Die erzeugte *CalcWrapper*-Instanz erlaubt das Aufrufen der Methoden zum Setzen und Auslesen von Eigenschaftswerten eines Prozesselements, ohne das identifizierende Kürzel der Eigenschaft angeben zu müssen.

```
1  class CalcWrapper:
2      def __init__(self, calc)
3          self._calc = calc
4
5          for i in range(len(self._calc.findGtProperties())):
6              prop = self._calc.findGtProperties()[i]
7              if prop.ident():
8
9                  funcName = re.sub('[^A-Za-z0-9]+', '', prop.ident())
10                 tempLetter = funcName[0]
11                 funcName = funcName.replace(tempLetter, tempLetter.lower(), 1)
12
13                 def dynGetter(self = self, propName = prop.ident()):
14                     return self._calc.propertyValue(propName)
15                 setattr(self, funcName, dynGetter)
16
17                 tempLetter = funcName[0]
18                 funcName = funcName.replace(tempLetter, tempLetter.upper(), 1)
19                 funcName = 'set' + funcName
20
21                 def dynSetter(val, self = self, propName = prop.ident()):
22                     self._calc.setPropertyValue(propName, val)
23                 setattr(self, funcName, dynSetter)
```

Abbildung 5.5
Ausschnitt der Konstruktordefinition der *CalcWrapper*-Klasse

```
calc.setExampleProperty(42)

propVal = calc.exampleProperty()

print (prop) #output: 42
```

Abbildung 5.6
Methoden einer *CalcWrapper*-Instanz zum Setzen und Auslesen von Eigenschaftswerten

In der Wrapper-Klasse wurden weitere Python-interne Funktionen überschrieben, die den Zugriff auf alle verfügbaren Parameter und Funktionen des umhüllten Prozesselements ermöglichen (siehe Abbildung 5.7). Dadurch kann die *CalcWrapper*-Instanz wie eine Instanz des umhüllten Prozesselements behandelt werden. Sie erweitert die Prozesselement-Instanz lediglich um Methoden zum Setzen und Auslesen der registrierten Eigenschaftswerte. Um das beschriebene Verhalten zu ermöglichen, mussten die Methoden *__getattr__()* und *__setattr__()* überschrieben werden. Die *__getattr__()*-Methode wird aufgerufen wenn ein unbekanntes Attribut der *CalcWrapper*-Instanz angefordert wird. Ein Beispiel dafür wäre der Methodenaufruf einer Methode, die nicht als Teil der Wrapper-Instanz definiert ist [7]. Sollte solch ein Fall auftreten, wird nach einem entsprechenden Attribut der Prozesselement-Instanz gesucht. Darüber ermöglicht die Wrapper-Instanz das Aufrufen von Methoden und Parametern des Prozesselements. Die *__setattr__()*-Methode wird dazu genutzt, das Verhalten bei der Zuweisung von Attributen zu bestimmen. Wird diese Methode überschrieben, werden Attributzuweisungen der

```
    def __getattr__(self, name):
        return getattr(self.__dict__['_calc'], name)

    def __setattr__(self, name, value):
        if name is ('_calc') or hasattr(self.__dict__['_calc'], name) is False:
            self.__dict__[name] = value
        else:
            setattr(self.__dict__['_calc'], name, value)

    def __dir__(self):
        return sorted(list(self.__dict__.keys()) + dir(self._calc))
```

Abbildung 5.7
Überschriebene Python-Methoden in der *CalcWrapper*-Klasse

Form `self.attr = value` als Methodenaufrufe der Form `self.__setattr__('attr', value)` interpretiert [7]. In der Wrapper-Klasse wurde diese Methode so überschrieben, dass überprüft wird, ob es sich um eine Zuweisung der Prozesselementen-Instanz handelt. Sollte dies der Fall sein, wird das Attribut der *CalcWrapper*-Instanz zugeordnet. Sollte jedoch ein Attribut zugewiesen werden, welches innerhalb der Prozesselement-Instanz definiert ist, wird die Attributzuweisung an das Prozesselement weitergeleitet. Die Methode *__dir__()* gibt eine Liste der Namen aller aufrufbaren Attribute einer Instanz zurück [16]. Die standardmäßige Implementierung würde eine Liste der Attribute der *CalcWrapper*-Instanz zurück geben. Durch das Überschreiben der *__dir__()*-Methode wurde erreicht, dass die zurückgegebene Liste die Namen aller Attribute der *CalcWrapper*- und der Prozesselement-Instanz enthält. In Anhang A befindet sich der Programmcode der gesamten Klassendefinition der *CalcWrapper*-Klasse.

In Abbildung 5.8 ist ein Beispiel für die Nutzung der Funktionalitäten dargestellt, deren Implementierungen in diesem Unterkapitel beschrieben wurden. Es wird die Methode zur Erzeugung eines beispielhaften Prozesselements aufgerufen. Sie gibt eine *CalcWrapper*-Instanz zurück, die das erzeugte Prozesselement umhüllt. Daraufhin wird ein Eigenschaftswert über eine Methode der *CalcWrapper*-Instanz festgelegt. Zur Ausführung des Prozesselements wird die *run()*-Methode aufgerufen. Diese Methode ist als Teil des Prozesselements implementiert.

```
1 exampleCalc = GtExampleCalculator('ExampleCalculator')
2
3 exampleCalc.setExampleProperty(42)
4
5 exampleCalc.run()
```

Abbildung 5.8
Beispielprogrammcode zur Instanziierung und Ausführung von Prozesselementen

5.2.2 Helferklassen der Prozesselemente im Python-Kontext

Das Eigenschaftssystem von GTlab ist auf die Verwaltung statischen Datenstrukturen ausgelegt. Die Eigenschaften einer *GtObject*-Unterklasse müssen bei der Implementierung festgelegt werden. Zur Laufzeit besteht keine Möglichkeit die Anzahl der Eigenschaftswerte dynamisch zu verändern. In verschiedenen Fällen werden dynamische Datenstrukturen zur Konfiguration von Prozesselementen benötigt. In GTlab werden dazu Helferklassen eingesetzt. Variiert die Anzahl der Eigenschaften in Abhängigkeit einer Benutzereingabe, kann einem Prozesselement eine entsprechende Anzahl der Instanzen einer Helferklasse untergeordnet werden. Jede Helferklassen-Instanz beinhaltet die Daten der dynamischen Eigenschaften. Dabei können Helferklassen ebenfalls die dynamische Datenstrukturen in Form von untergeordneten Helferklassen verwenden. Für die skriptbasierte Prozesssteuerung mussten die Helferklassen innerhalb des Python-Kontextes instanziierbar werden. Dazu wurde eine weitere Factory-Klasse erstellt, die das Erzeugen von Instanzen der Helferklassen ermöglicht (siehe Abbildung 5.9). Innerhalb der Factory-Insatnz können Helferklassen registriert werden. Dabei wird die *QMetaObject*-Instanz der Helferklasse in der Factory-Instanz gespeichert. Die Factory-Instanz ermöglicht zudem das Verwalten

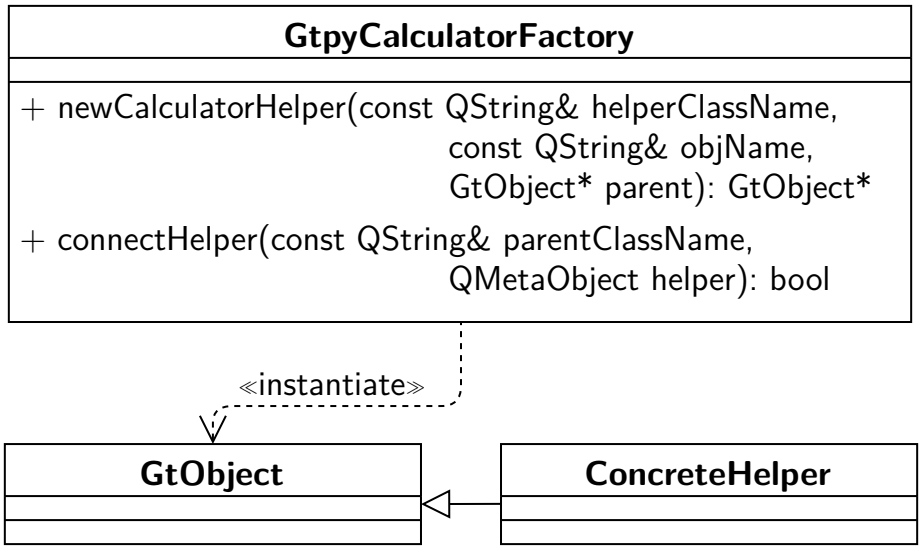

Abbildung 5.9
Implementierung der Factory-Klasse zur Erzeugung von Helferklassen-Instanzen

von Beziehungen zwischen Helferklassen und ihren möglichen Elternklassen. Dazu kann bei der Registrierung einer Helferklasse der Klassenname der Elternklasse angegebene werden. Die Factory-Instanz verwaltet somit die Information, welchen Elternklassen die Helferklassen untergeordnet werden können. Zur Erzeugung einer Helferklassen-Instanz muss der Factory-Instanz der Klassenname der entsprechenden Helferklasse übermittelt werden. Die Factory-Instanz erzeugt daraufhin ein Objekt der Helferklasse und hängt dieses an das übergebene Elternelement an. Um den Zugriff auf die Helferklassen innerhalb des Python-Kontext zu ermöglichen, wurde die Factory-Instanz in den Python-Kontext integriert.

Helferklassen lassen sich über die Factory-Instanz eindeutig zu den möglichen Elternklassen zuordnen. Auf Grundlage dessen, konnte der Konstruktor der *CalcWrapper*-Klasse um den in Abbildung 5.10 dargestellten Python-Programmcode erweitert werden. Beim Erzeugen einer *CalcWrapper*-Instanz wird eine Liste der Namen aller Helferklassen angefordert, die dem umhüllten Prozesselement untergeordnet werden können. Für jede mögliche Helferklasse wird die *CalcWrapper*-Instanz um eine Methode erweitert. Die Methode erzeugt unter Nutzung der Factory-Instanz eine Instanz der Helferklasse und hängt sie an das Prozesselement an. Die Helferklassen-Instanz wird ähnlich wie die Prozesselemente von einer Wrapper-Klasse namens *HelperWrapper* umhüllt. Auf diese Weise können die Helferklassen-Instanzen ebenfalls mit zusätzlichen Methoden ausgestattet werden. Das ist notwendig, da einer Helferklassen-Instanz weitere Helferklassen-Instanzen untergeordnet sein können. Die *HelperWrapper*-Klasse erweitert die Helferklassen-Instanz um Methoden zum Erzeugen und Anhängen der möglichen Kindelemente. In Anhang B ist der Python-Programmcode der *HelperWrapper*-Klasse abgebildet.

5.3 Erstellung eines skriptgesteuerten Python-Tasks

Aus der in Abschnitt 4.3 beschrieben Analyse der Prozesssteuerung von GTlab ging hervor, dass Prozessketten innerhalb eines eigenen Threads ausgeführt werden. Dabei

```
25        helpers = HelperFactory.connectedHelper(self._calc.__class__.__name__)
26        for i in range(len(helpers)):
27            funcName = 'create' + helpers[i]
28
29            def dynCreateHelper(name, self = self, helperName = helpers[i]):
30                helper = HelperFactory.newCalculatorHelper(helperName, name, self.
                      _calc)
31                return HelperWrapper(helper)
32            setattr(self, funcName, dynCreateHelper)
```

Abbildung 5.10
Erweiterung der Konstruktorimplementierung der *CalcWrapper*-Klasse zum Erstellen von Helferinstanzen

initiiert der Wurzeltask die sequenzielle Ausführung der untergeordneten Prozesskomponenten. Die skriptbasierte Prozesssteuerung muss genau an dieser Stelle eingreifen und die im Skript definierte Routine zur Ausführung der Prozesskomponenten zwischenschalten. Dazu kann ein Task erstellt werden, der anstelle der sequenziellen Ausführung einen frei definierbaren Python-Code an einen Python-Interpreter übergibt und ausführen lässt. Für den Python-Interpreter muss es möglich sein, Prozesselemente zu erzeugen und auszuführen.

Zur Erstellung der skriptbasierten Prozesssteuerung wurde innerhalb des Python-Moduls die Klasse *GtpyTask* angelegt, die von der *GtTask*-Klasse abgeleitet ist (siehe Abbildung 5.11). Die *GtpyTask*-Klasse stellt einen neuen Task dar, der über eine Modul-Schnittstelle in GTlab integriert werden kann. Um das Ausführungsverhalten des Tasks zu definieren, wurde die virtuelle Methode *runIteration()* überschrieben. Die *runIteration()*-Methode wurde so implementiert, dass sie unter Nutzung der Python-Schnittstelle eine Kopie des zentralen Datenmodells an den neu erstellten Python-Kontext übergibt. Daraufhin wird der Python-Code aus der klasseninternen Skriptvariable innerhalb des Python-Kontexts ausgeführt. Die Skriptvariable stammt vom GTlab-eigenen Datentyp *GtStringProperty* ab und wurde als Eigenschaftswert des Tasks registriert. Durch die Erstellung des Python-Tasks wurde es möglich Prozessketten mit Hilfe von Python-Code zu definieren und in der Skriptvariable zu hinterlegen. Bei der Ausführung des Python-Tasks über die konventionelle Prozesssteuerung wird die im Python-Code definierte Prozess-

kette ausgeführt. Über den Python-Task wurde die skriptbasierte Prozesssteuerung in GTlab integriert.

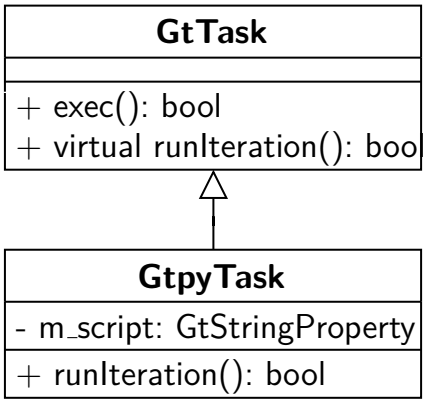

Abbildung 5.11
Implementierung des Python-Tasks

5.4 Grafische Benutzeroberfläche zur Definition von Python-Prozessketten

Zur Definition einer Prozesskette mit Hilfe von Python-Code wurde eine grafische Benutzeroberfläche erstellt. In GTlab steht eine Basisklasse zur Implementierung von Benutzeroberflächen für Prozesskomponenten bereit. Die individuelle gestaltbare Benutzeroberfläche dient zur Konfiguration der registrierten Eigenschaften einer Prozesskomponente. Bereits vor der Durchführung dieser Arbeit existierte ein Prozesselement, zur Definition von Berechnungsverfahren in Form von Python-Skripten [17]. Das Prozesselement ist mit einer Benutzeroberfläche ausgestattet, die aus einem Editor und einer Ausgabekonsole besteht (siehe Abbildung 5.12). Im Editor steht eine automatische Vervollständigung zur Verfügung, die als Unterstützung bei der Erstellung von Python-Skripten dient.

Die grafische Benutzeroberfläche des Python-Tasks musste ebenfalls einen Editor bereitstellen, der das Definieren von Python-Skripten ermöglicht. Im Rahmen der vorliegenden

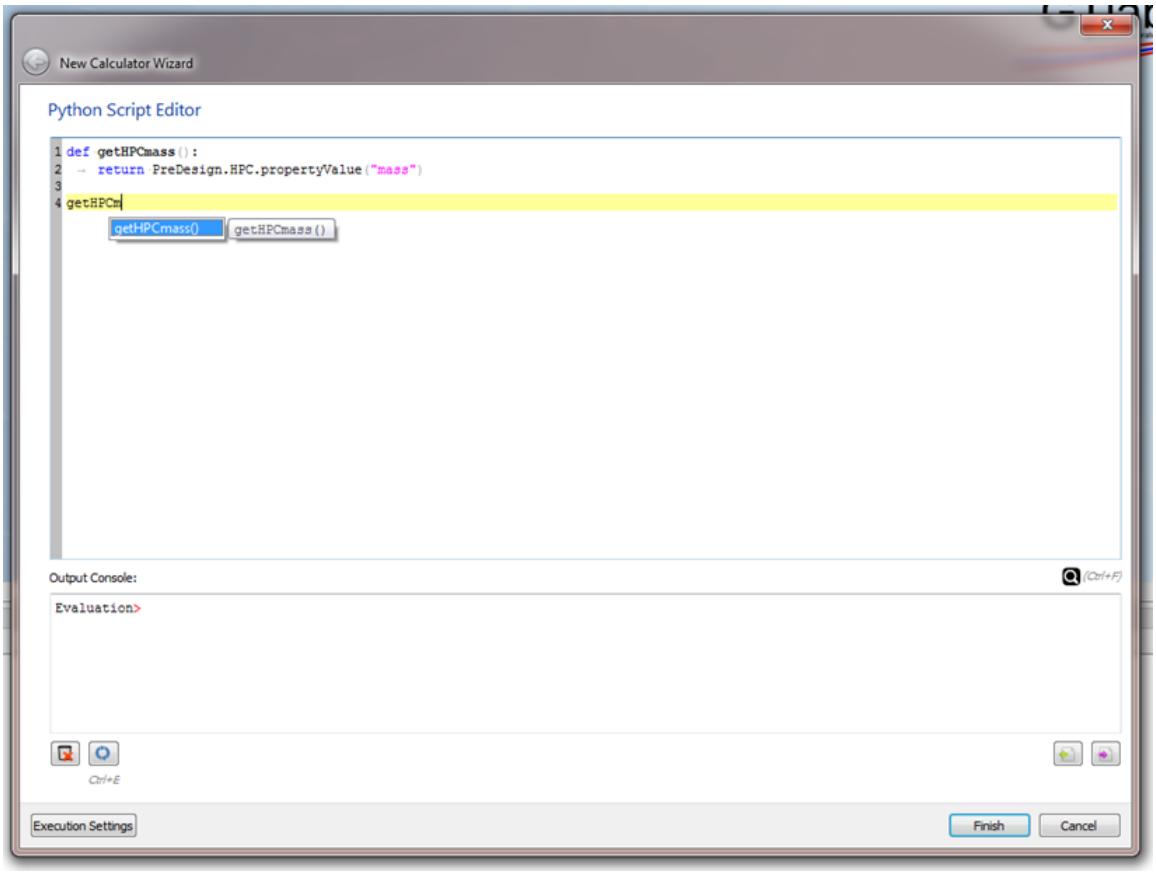

Abbildung 5.12
Grafische Benutzeroberfläche des Python-Prozesselements

Arbeit wurde eine Abstraktionsstufe eingeführt, die den grundlegenden Aufbau der Benutzeroberfläche zum Definieren von Python-Skripten beinhaltet (siehe Abbildung 5.13). Dazu wurde die bereits bestehenden Implementierungen von Editor und Ausgabekonsole in die abstrakte Klasse verschoben. Die abstrakte Klasse leitet von der Basisklasse zur Erstellung von Benutzeroberflächen für Prozesskomponenten ab. Somit konnten die Benutzeroberflächen des Python-Prozesselements und des Python-Task auf Grundlage der abstrakten Klasse erstellt werden, was die mehrfache Implementierung gleicher Programmteile verhindert. Im Folgenden werden die Funktionalitäten und die Verwendung der neu erstellten, abstrakten Klasse erläutert.

Die abstrakte Klasse *GtpyAbstractScriptingWizardPage* beinhaltet die Implementierung

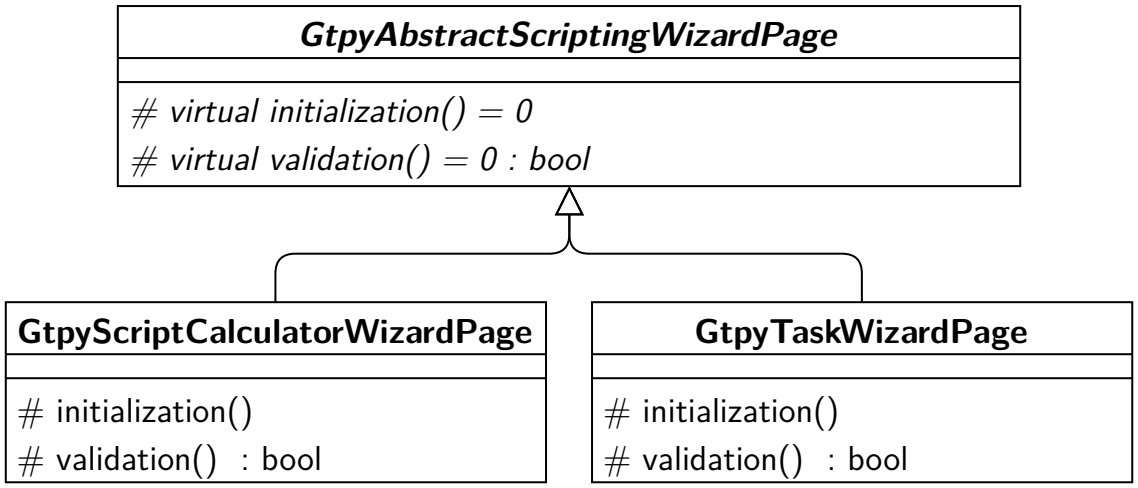

Abbildung 5.13
Abstraktion der Benutzeroberfläche zum Erstellen von Python-Code

der grafischen Benutzeroberfläche (siehe Abbildung 5.12). Dabei bietet die abstrakte Klasse die Möglichkeit, die Benutzeroberfläche innerhalb einer konkreten Implementierung um zusätzliche Elemente zu erweitern. Die zusätzlichen Elemente lassen sich neben dem Editor platzieren. Zudem deklariert die abstrakte Klasse die rein virtuellen Methoden *initialization()* und *validation()*. Unter Verwendung der Funktionalitäten der Basisklasse, ruft die abstrakte Klasse die *initialization()*-Methode unmittelbar vor der Darstellung der Benutzeroberfläche auf. Innerhalb dieser Methode kann das Python-Skript aus der Prozesskomponente ausgelesen und im Editor der Benutzeroberfläche platziert werden. Der Aufruf der *validation()*-Methode findet hingegen unmittelbar vor dem Schließen der Benutzeroberfläche statt. Dabei kann das möglicherweise bearbeitete Skript aus dem Editor ausgelesen und in der Prozesskomponenten gespeichert werden. Dieses Verhalten muss in jeder konkreten Implementierung der abstrakten Klasse individuell definiert werden, da die Art des Zugriffs auf die Prozesskomponente davon abhängt, ob es sich einen Task oder ein Prozesselement handelt. Innerhalb der abstrakten Klasse wird die Python-Schnittstelle zur Ausführung von Python-Code angesprochen. Darüber ist die Evaluierung des Python-Codes im Editor möglich. In der konkreten Implementierung muss der Python-Kontext für die Evaluierung des Python-Codes bestimmt werden. Alle verfügbaren Objekte, Methoden und Klassen, die in dem gewählten Kontext zur

Verfügung stehen, werden über die automatische Vervollständigung im Editor vorgeschlagen. Durch die Evaluierung können weitere benutzerdefinierte Komponenten im Python-Kontext und somit in die automatische Vervollständigung registriert werden. Dabei besteht der Zugriff auf eine Kopie des zentralen Datenmodells. Es ist zu beachten, dass die Modifikationen an der Kopie des Datenmodells durch die Evaluierung nicht ins zentrale Datenmodell zurückgeführt werden. Zudem werden die erstellten Prozesselemente bei der Evaluierung nicht ausgeführt. Besonders komplexe Prozesselemente könnten anderenfalls die Dauer der Evaluierung negativ beeinflussen.

Zur Erstellung der Benutzeroberfläche des Python-Prozesselements wurde die Klasse *GtpyScriptCalculatorWizardPage* angelegt. In dieser Klasse wurden die Methoden *initialization()* und *validation()* auf die zuvor beschriebene Weise implementiert. Sie nutzten dabei die Möglichkeit auf das Python-Prozesselement zuzugreifen, welches zur Konfiguration ausgewählt wurde. Es mussten keine weiteren Anpassungen vorgenommen werden, da der benötigte Aufbau der Benutzeroberfläche bereits in der abstrakten Klasse definiert ist.

Die Benutzeroberfläche des Python-Tasks wurde in der Klasse *GtpyTaskWizardPage* implementiert. Die *GtpyTaskWizardPage*-Klasse greift innerhalb der Methoden *initialization()* und *validation()* auf den entsprechenden Python-Task zu, um das darin gespeicherte Skript bearbeiten zu können. Es wurde die Möglichkeit genutzt, die vordefinierte Benutzeroberfläche der abstrakten Klasse zu erweitern (siehe Abbildung 5.14). Dabei wurde neben dem Editor eine Auflistung aller im Python-Code instanziierten Prozesselemente platziert. Über die automatischen Vervollständigung können die vordefinierten Methoden zur Erstellung von Prozesselementen im Editor aufgerufen werden. Zudem bietet die Benutzeroberfläche eine grafische Unterstützung zur Erstellung von Prozesselementen an. Über die Schaltfläche oberhalb der Auflistung lässt sich um eine Ansicht aller Verfügbaren Prozesselemente von GTlab einsehen. Nach der Auswahl eines Prozesselements wird dessen individuell definierte Benutzeroberfläche zur Konfiguration der Eigenschaftswerte angezeigt. Die Konfiguration in der Benutzeroberfläche wird automatisiert in Python-Code überführt und im Editor eingefügt. Ein Anwender kann auf diese

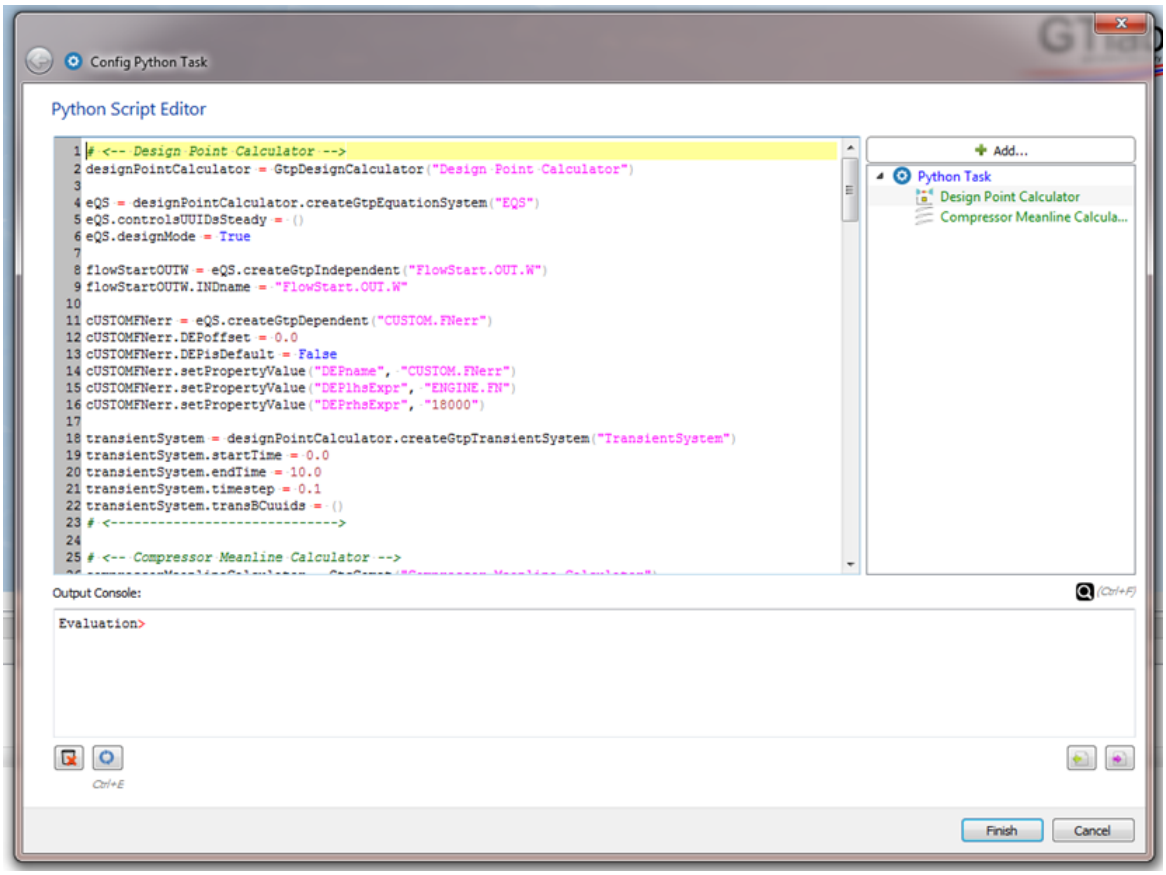

Abbildung 5.14
Grafische Benutzeroberfläche des Python-Tasks

Weise die gewohnte Benutzeroberfläche zur Konfiguration von Prozesselementen verwenden und muss keine Spezialkenntnisse über den dazu benötigten Python-Programmcode vorweisen. Die Auflistung ermöglicht das erneute Konfigurieren der im Python-Code vorhandenen Prozesselemente. Der automatisiert erzeugte Python-Programmcode wird daraufhin im Bezug auf die veränderte Konfiguration angepasst.

6 Skriptgesteuerte Abschätzung der Masse einer Triebwerkskomponente

Die Funktionalität, der im Rahmen der vorliegenden Arbeit erstellten, skriptbasierten Steuerung der Prozessketten, wurde an einem adaptierten Anwendungsfall demonstriert. Kern der Anwendung war eine Massenabschätzung einer einzelnen Triebwerkskomponente. Um ein generelles Verständnis über den Anwendungsfall zu bekommen, wird zunächst der allgemeine Aufbau eines Triebwerks erläutert und dessen Arbeitsweise anhand der Darstellung in Abbildung 6.1 beschrieben. Es ist zu erkennen, dass sich ein Triebwerk im laufenden Betrieb in die vier Funktionsbereiche Ansaugen, Verdichten, Verbrennen und Ausstoßen unterteilen lässt. Im ersten Bereich wird die Umgebungsluft durch den Lufteinlass in das Triebwerk eingesaugt. Die Luft wird in einem Verdichter komprimiert und anschließend in einer Brennkammer zusammen mit dem Brennstoff

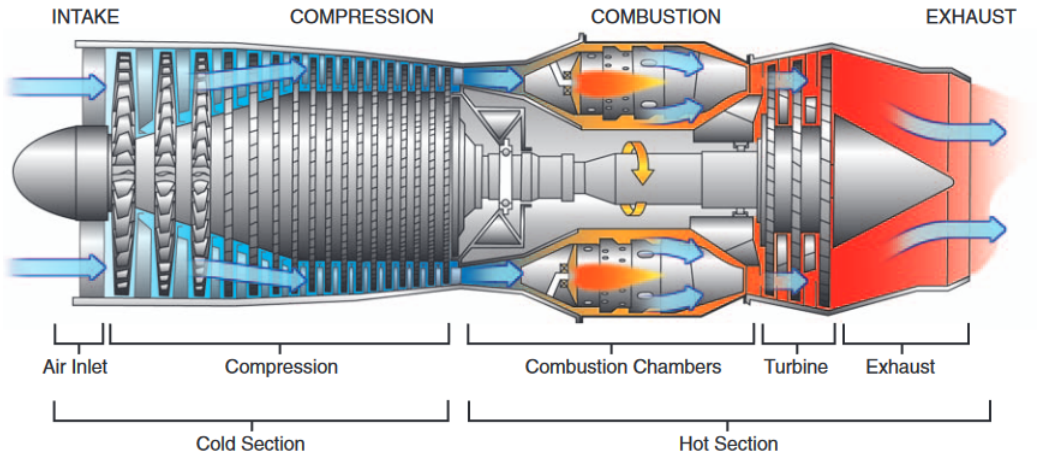

Abbildung 6.1
Allgemeiner Aufbau eines Triebwerks [18]

verbrannt. Aufgrund des Temperaturanstiegs expandiert das Gasgemisch, wodurch sich dessen Strömungsgeschwindigkeit erhöht. Der Gasstrom wird durch die Turbine geleitet, die ein Teil der thermische Energie des Gases in mechanische Arbeit umwandelt. Die Turbine ist über eine Welle mit dem Verdichter verbunden und treibt diesen an. Der Ausstoß des beschleunigten Gases erzeugt einen Schub, der für den Antrieb eines Flugzeugs eingesetzt werden kann. Anders als bei einer Kolbenmaschine, erfolgen die Arbeitsschritte in der betrachteten Strömungsmaschine kontinuierlich. Das hat zur Folge, dass zu jedem Zeitpunkt im laufenden Betrieb Brennstoff verbrannt und somit chemische in kinetische Energie umgewandelt wird.

Der Anwendungsfall für die skriptbasierte Prozessteuerung konzentriert sich auf einen Axialverdichter eines Triebwerks vom Typ V2500. Solche Triebwerke kommen bei Passagierflugzeugen für die Kurz- und Mittelstrecken zum Einsatz. Bei einem Axialverdichter handelt es sich um eine Triebwerkskomponente, die die einströmende Luft in axialer Richtung verdichtet. Der luftdurchströmte Raum eines Axialverdichters wird als Ringraum bezeichnet. Aus Abbildung 6.1 geht hervor, das die Kompression im Ringraum über mehrere Stufen vollzogen wird, wobei jede Stufe eine Druckerhöhung bewirkt. Eine Verdichterstufe besteht aus einer Reihe Rotorschaufeln gefolgt von einer Reihe Statorschaufeln. Die Rotorschaufeln werden über die Welle, die mit der Turbine verbunden ist, in Rotation versetzt und beschleunigen somit die durchströmende Luft. Die Statorschaufeln sind fest mit dem Gehäuse des Verdichters verbunden. Dadurch verlangsamen sie den Luftstrom und verhindern dessen Verwirbelung. Sie bestimmen den Winkel, in welchem die Luft zur nächsten Stufe weitergeleitet wird. Die Verlangsamung des Luftstroms durch die Statorschaufeln bewirkt die Kompression der Luft [19].

Triebwerksauslegungen mit einer Variation der Schubkraft, haben eine Variation der Komponentengeometrien und somit auch der Komponentenmassen zur Folge. Im vorliegenden Anwendungsfall wurde die Masse einer Verdichterkomponente bei gleichbleibenden Flugbedingungen und unterschiedlichen Schubanforderungen ans Triebwerk abgeschätzt. Als Werkzeug zur Massenabschätzung wurde das Programmsystem GTlab und die skriptbasierte Prozesssteuerung verwendet. Es wurde eine Prozesskette in Form eines

Python-Skripts adaptiert, welche die Abschätzung der Masse des Verdichters iterativ und unter Berücksichtigung der steigenden Schubanforderung durchführt.

6.1 Vorgehensweise zur Massenabschätzung eines Axialverdichters

Um eine Abschätzung der Masse einer einzelnen Triebwerkskomponente durchzuführen, muss der thermodynamische Kreisprozess des Gesamttriebwerks im Flugbetrieb berechnet werden. Der Kreisprozess beschreibt die Zustandsänderung des Fluids, welches das Triebwerk durchströmt. Daraus ergeben sich unter anderem Erkenntnisse über Druck und Temperatur an den einzelnen Stationen des Triebwerks. Um den thermodynamischen Kreisprozess mit Hilfe von GTlab zu berechnen, muss als initialer Schritt der Gesamtaufbau des betrachteten Triebwerks definiert werden. Das Modul *Performance* ermöglicht es Triebwerkskomponenten, über die grafische Benutzeroberfläche von GTlab, miteinander zu verknüpfen und somit ein Triebwerkmodell zu definieren. Unter Verwendung der Berechnungsverfahren, die das *Performance*-Modul bereitstellt, lässt sich für den definierten Triebwerksaufbau und unter Verwendung weniger Eingabeparameter der thermodynamische Kreisprozess des Triebwerks während des Betriebs ermitteln. Der berechnete Kreisprozess kann zur konzeptionellen Abschätzung der Eigenschaften einzelner Triebwerkskomponenten herangezogen werden. Dazu stellt das Modul *Sketchpad* Berechnungsverfahren bereit, die eine konzeptionelle Abschätzung von Aerodynamik und Strukturmechanik einzelner Triebwerkskomponenten ermöglichen. Aus der konzeptionellen Abschätzung gehen erste Eigenschaften der Triebwerkskomponenten, wie zum Beispiel Masse und Abmessung, hervor. Durch das Zusammenspiel von *Performance* und *Sketchpad*, lässt sich die Auswirkung des thermodynamischen Kreisprozesses auf die Geometrie und die Masse der einzelnen Triebwerkskomponenten bewerten.

In Abbildung 6.2 ist der Triebwerksaufbau des V2500 Triebwerks dargestellt, wie er unter Nutzung des *Performance*-Moduls für die in diesem Kapitel beschriebene Massenabschätzung definiert wurde. Die Abschätzung bezieht sich auf den Axialverdichter,

welcher in der Abbildung mit dem Kürzel *HPC* gekennzeichnet ist. Um die Masse des Axialverdichters abzuschätzen, muss der thermodynamische Kreisprozess des dargestellten Triebwerks berechnet werden. Der Kreisprozess verändert sich, sobald sich die Schubanforderungen an das Triebwerk verändern. Die Berechnung ist so konfiguriert, dass die Schuberhöhung durch die Anpassung des Massenstroms des durchströmenden Fluids erreicht wird. Der Temperatur- und Druckverlauf, den das Fluid beim Durchströmen des Triebwerks erfährt, wird bei jeder Iteration konstant gehalten. Wurde der Kreisprozess innerhalb einer Iteration berechnet, kann dessen Ergebnis für die aerodynamische Auslegung des Axialverdichters herangezogen werden. Für diese Berechnung wurde definiert, dass der Verdichter aus acht Stufen bestehen soll. Unter Verwendung der Werte für Druck, Temperatur und Massenstrom am Eintritt des Verdichters, welche aus dem Kreisprozess hervorgehen, kann eine konzeptionelle Geometrie des Ringraums und der darin platzierten Schaufeln berechnet werden. Es ist ein geeignetes Material auszuwählen, aus dem die Schaufeln bestehen können. Anhand der Geometriedaten und der Dichte des gewählten Materials lässt sich die Masse der einzelnen Schaufeln bestimmen. Die Masse der Rotorschaufeln steht im direkten Zusammenhang mit der Kraft, die durch die Rotation im laufenden Betrieb auf die Rotorschaufeln wirkt. Die Füße der Rotorschaufeln müssen dieser Kraft entgegenwirken. Sie dienen zur Verbindung der Schaufeln mit den Scheiben, die über die Welle durch die Turbine angetrieben werden. Die Geometrien und Massen der Schaufelfüße und der Scheiben, lassen sich unter Berücksichtigung der zuvor berechneten Schaufelmassen ermitteln [20]. Sollte es dazu kommen, dass sich eine Rotorschaufel im laufenden Betrieb löst, muss die

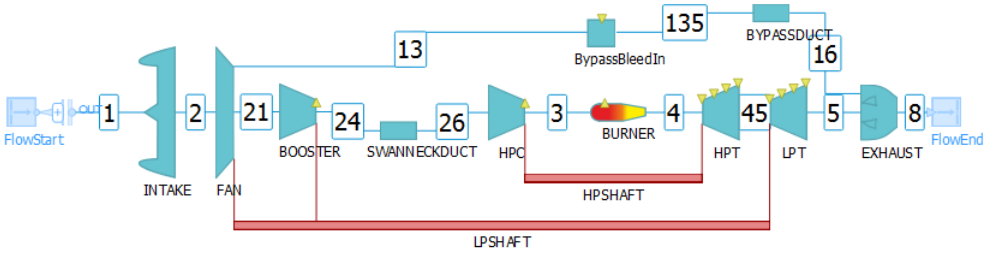

Abbildung 6.2
Triebwerkmodell eines V2500 Triebwerks im *Performance*-Modul von GTlab

Kraft durch ein ausreichend dickes Gehäuse des Verdichters abgefangen werden. Die benötigte dicke des Gehäuses an den einzelnen Stufenabschnitte können ebenfalls unter Berücksichtigung der Schaufelmassen berechnet werden. Letztlich müssen die Massen der einzelnen Komponenten addiert werden, um die Gesamtmasse des Verdichters zu bestimmen. Durch die beschriebene Parameterstudie können die Gesamtmassen der Verdichter abgeschätzt werden, die für die unterschiedliche Schubanforderungen bei gleichen Flugbedingungen in dem Triebwerk verbaut sein müssten.

6.2 Massenabschätzung eines Axialverdichters unter Verwendung der Python-Prozesskette

Zur Durchführung der zuvor beschriebenen Berechnungen wurde die im Rahmen dieser Arbeit implementierte Python-Prozesskette verwendet. Dessen Benutzeroberfläche ermöglicht es Python-Skripte über einen Editor zu erstellen. Innerhalb der Skripte können Prozesselemente instanziiert und konfiguriert werden. Der Python-Programmcode in Abbildung 6.3 führt eine iterative Massenabschätzung des Verdichters eines V2500 Triebwerks für unterschiedliche Schubanforderungen aus. Aus Gründen der Übersicht, wurde darauf verzichtet die Definitionen aller verwendeten Methoden abzubilden. Es wird exemplarisch dargestellt, dass diese die Konfigurationen der benötigten Prozesselemente beinhalten. Zur Abschätzung der Masse des Verdichters wurde zunächst eine Liste erzeugt, die die Werte der betrachteten Schubanforderungen enthält. Die Werte wurden in der Einheit Newton (N) angegeben und beziehen sich auf die Schubkraft des Triebwerks. Sie bestimmen die Leistungsfähigkeit des betrachteten Triebwerks, die mit jedem Listeneintrag um 2000 N erhöht wird. Die Anzahl der Listeneinträge bestimmt die Anzahl der Parameterstudien und somit die Anzahl der Verdichter dessen Masse abgeschätzt wird. Zu Beginn jeder Iteration wird ein Wert für den Schub aus der Liste ausgelesen und ein Name für den Datensatz bestimmt, der während einer Parameterstudie erzeugt wird. Ein Datensatz beinhaltet die konzeptionellen Vorauslegungsdaten eines Verdichters. Daraufhin wird die Berechnung der Performance ausgeführt. Die Ergebnisdaten für den Kreisprozess werden in die darauffolgende Berechnung der Aerodynamik des Verdichters

einbezogen. Daraus entstehen erste Abschätzungen der Geometrie des Ringraums und der Schaufelreihen. Im weiteren Auslegungsverlauf erfolgt die Berechnung der Schaufelmassen, dessen Wert für die Auslegung der Schaufelfüße und der Scheibe herangezogen wird. Zudem wird die Wandstärke jeder Stufe des Verdichters berechnet. Im finalen Schritt erfolgt die Addition der berechneten Einzelmassen, um die Gesamtmasse des Verdichters zu bestimmen.

Das Anwendungsbeispiel verdeutlicht die Flexibilität im Aufbau von Prozessketten, die durch die skriptbasierte Prozesssteuerung in GTlab integriert wurde. Prozesselemente lassen sich zukünftig in flexibler Reihenfolge, beliebig oft und in Abhängigkeit von berechneten Ergebnisdaten ausführen. Dabei können Daten aus dem zentralen Datenmodell

```
335 # <-- hpcProperties -->
336 diskPropertiesCalculator = GtsDiskPropertiesCalculator("hpcProperties")
337 diskPropertiesCalculator.setMat("Inconel 718")
338 # <-------------------->
339
340 def runHPCproperties(name):
341     diskPropertiesCalculator.setTarget(PreDesign.findGtChild(name))
342     diskPropertiesCalculator.run()
343
344 FNvals = [18000, 20000, 22000, 24000, 26000]
345
346 for i in range(0, len(FNvals)):
347
348     fnval = FNvals[i]
349     hpcName = "HPC_FN_" + str(fnval)
350
351     perfDatDesign = runPerf(fnval)
352
353     runHPCcomet(perfDatDesign, hpcName)
354
355     runHPCbladeMass(hpcName)
356     runHPCbladeFoot(hpcName)
357     runHPCdisks(hpcName)
358     runHPCcasing(hpcName)
359     runHPCproperties(hpcName)
360
361     calcHPCmass(hpcName)
```

Abbildung 6.3
Python-Programmcode zur iterativen Massenabschätzung eines Verdichters in GTlab

ausgelesen und modifiziert werden. Im Anwendungsfall konnte die Massenabschätzung iterativ ausgeführt werden, so dass die Massen der verschiedenen Verdichterauslegungen zum direkten Vergleich zur Verfügung standen. Nach jeder Iteration wurden die Massen der einzelnen Komponenten des Verdichters addiert um die Gesamtmasse des Verdichters zu bestimmen und ins Datenmodell zu überführen. Für die Berechnung der Gesamtmasse steht kein Prozesselement in GTlab bereit, weshalb dies über den Python-Programmcode in Abbildung 6.4 realisiert wurde. Darin werden die Schaufelreihen eines Verdichterdatensatzes ausgelesen, um die Masse der Schaufeln einer Reihe zu ermitteln. Handelt es sich bei der Schaufelreihe um Rotorschaufeln, wird zudem die Masse der Schaufelfüße, sowie die der Scheibe zur Gesamtmasse des Verdichters addiert. Die berechnete Gesamtmasse konnte ins Datenmodell überführt werden. Daran wird ein weiterer Vorteil der skriptbasierten Prozesssteuerung deutlich. Die benötigte Berechnung konnte über Python-Programmcode direkt in GTlab implementiert werden. Es mussten keine Anpassungen am GTlab-Programmcode vorgenommen werden, die eine Kenntnis der GTlab-Schnittstellen vorausgesetzt und das erneute Kompilieren einzelner Programmteile notwendig gemacht hätten. So lassen sich zukünftig beliebige

```
257 def calcHPCmasses(name):
258     totalMass = 0
259     blades = PreDesign.findGtChild(name).Blades.findGtChildren()
260 
261     for i in range(len(blades)):
262         blade = blades[i]
263         nob = blade.propertyValue("nob")
264         bladeMass = blade.propertyValue("bladeMass")
265 
266         totalMass += nob * bladeMass
267 
268         if blades[i].className() == "GtdRotorBladeRow":
269             attMass = blade.Attachment.propertyValue("mass")
270             diskMass = blade.Disk.propertyValue("mass")
271 
272             totalMass += nob * attMass
273             totalMass += diskMass
274 
275     PreDesign.findGtChild(name).setPropertyValue("mass", totalMass)
```

Abbildung 6.4
Python-Programmcode zur Berechnung der Gesamtmasse eines Verdichters in GTlab

Berechnungen über Python-Programmcode direkt in GTlab integrieren. Zudem ist eine prototypische Entwicklung zukünftiger Erweiterungen von GTlab über die skriptbasierte Prozesssteuerung möglich. Dazu können Berechnungsverfahren innerhalb der Python-Umgebung getestet werden. Bewähren sich die Berechnungsverfahren, können sie in ein Prozesselement überführt und somit in GTlab integriert werden.

6.3 Ergebnis der Massenabschätzung des Verdichters

Die Geometriedaten der berechneten Verdichter konnten unter Verwendung des GTlab-Moduls *PreDesign* visualisiert werden. In Abbildung 6.5 ist die Visualisierung der Verdichtergeometrie dargestellt, die für ein V2500 Triebwerk mit der Schubanforderung von 18000 N abgeschätzt wurde. Es handelt sich um einen halben Querschnitt des Verdichters, der rotiert um die x-Achse den Aufbau des gesamten Verdichters wiedergibt. Die gestrichelte Umrandungen markieren eine Verdichterstufe bestehend aus einer Rotor- und einer Statorschaufelreihe. Die Rotorschaufeln sind über den Schaufelfuß mit der zugehörigen Scheibe verbunden. Die Statorschaufeln sind hingegen mit dem Verdichtergehäuse verbunden. Zur Auswertung der Massenabschätzung wurde die Ergebnisdaten in dem Koordinatensystem in Abbildung 6.6 visualisiert. Es wird deutlich, dass die Höhe der letzten Rotorschaufel bei steigender Schubanforderung zunimmt. Durch die veränderte Geometrie erhöht sich auch die Masse des Axialverdichters.

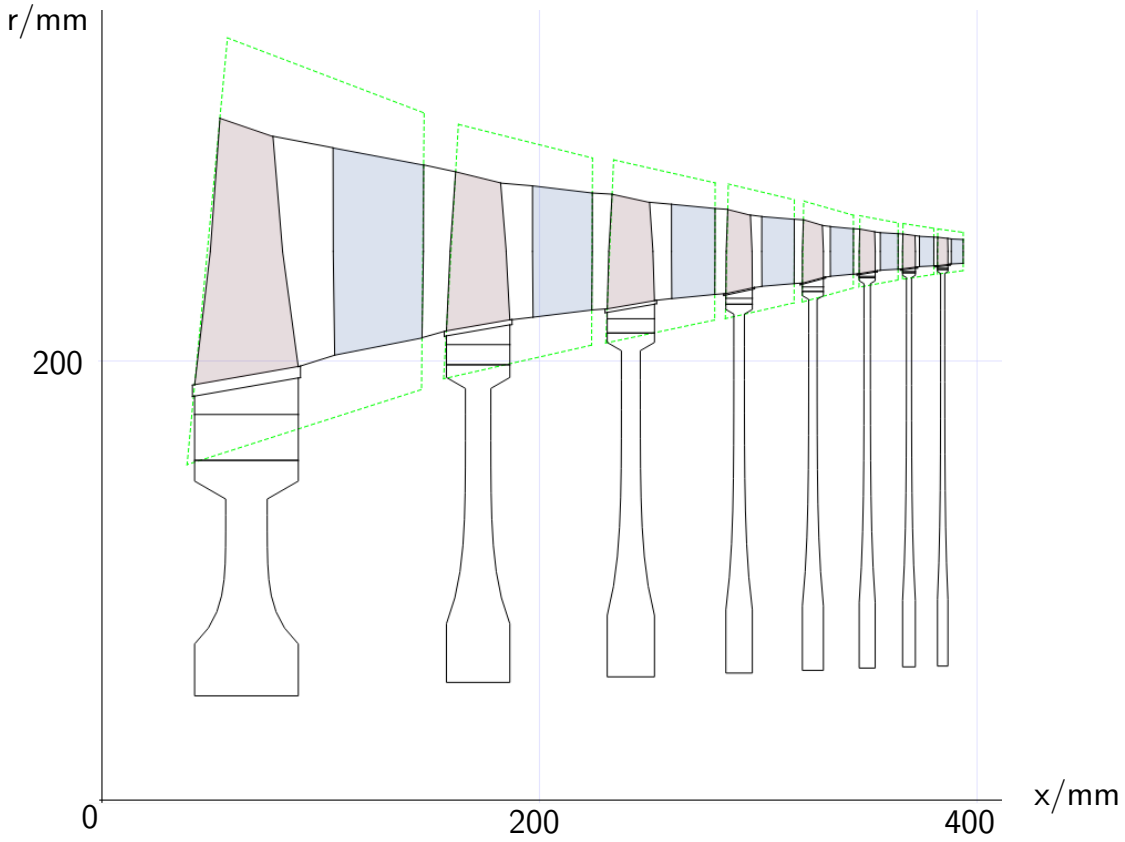

Abbildung 6.5
Geometrie eines Verdichters visualisiert mit *PreDesign*-Modul von GTlab

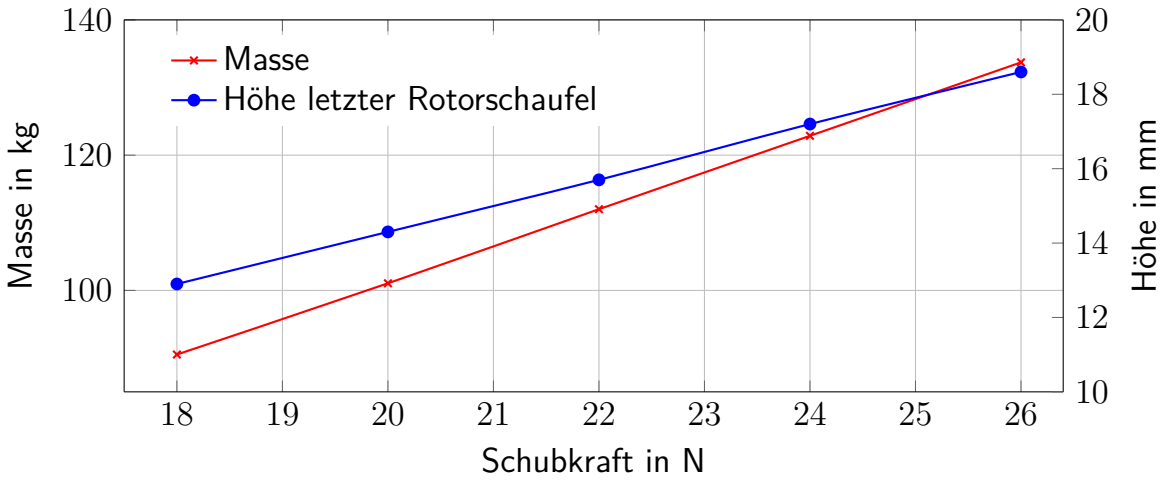

Abbildung 6.6
Ergebnisdaten der Massenabschätzung eines Axialverdichters für ein V2500 Triebwerk

7 Zusammenfassung und Ausblick

Ziel dieser Bachelorarbeit war die Integration einer skriptbasierten Prozesssteuerung in GTlab, die das Erstellen und Ausführen von Prozessketten in Form von Python-Skripten ermöglicht. Zum Einstieg in die Konzeptionierung der skriptbasierten Prozesssteuerung wurden das zentrale Datenmodell und die bestehende Prozessteuerung von GTlab analysiert. Dabei wurde deutlich, dass das zentrale Datenmodell das System Triebwerk virtuell abbildet und zum Austausch von Ergebnisdaten während eines Berechnungsprozesses dient. Die interne Prozesssteuerung von GTlab ist für die Ausführung von Prozessketten verantwortlich und ermöglicht dabei nahtlosen Zugriff auf das zentrale Datenmodell. Eine Prozesskette besteht aus einem Task, der das Ausführungsverhalten der Prozesskette beeinflusst und als Container für weitere Tasks und Prozesselemente fungiert. Innerhalb der Prozesselemente sind Berechnungsverfahren für den Triebwerkvorentwurf implementiert. Bei der Ausführung eines Tasks initiiert dieser die sequenzielle Ausführung der untergeordneten Komponenten. Anhand der Analyse der bestehenden Prozessteuerung wurde ein Konzept zur Integration einer skriptbasierten Prozesssteuerung in GTlab erstellt.

Zur Realisierung des Konzepts wurde die bereits vorhandene Python-Schnittstelle von GTlab verwendet. Über diese konnte ein Python-Interpreter zur Ausführung von Python-Code angesprochen werden. Die Python-Schnittstelle war fest in der Kernfunktionalität von GTlab verankert, wodurch eine feste Abhängigkeit zu Python bestand. Im Verlauf der vorliegenden Arbeit wurde die Python-Schnittstelle und somit die Abhängigkeit von Python in ein neu erstelltes GTlab-Modul überführt, welches als Python-Modul bezeichnet wird. Daraus ergibt sich der Vorteil, dass GTlab zukünftig unabhängig von Python ausgeführt werden kann. Das Python-Modul kann bei Bedarf dynamisch in GTlab integriert werden, so dass die Python-Funktionalitäten optional zur Verfügung stehen. Innerhalb des Python-Moduls wurde die Python-Schnittstelle um einen Kontext erwei-

tert, der als Umgebung zur Ausführung von Python-Code verwendet werden kann. Der Kontext wurde mit Methoden zum Erzeugen und Konfigurieren von Prozesselementen ausgestattet. Dadurch können Prozesselemente über Python-Code flexibel verschaltet und ausgeführt werden. Um diese Funktionalität in GTlab nutzen zu können, wurde das Python-Modul um einen Task erweitert. Anstelle der sequenziellen Ausführung untergeordneter Komponenten, führt der erstellte Task einen benutzerdefinierbaren Python-Code aus. Der Python-Code bestimmt dabei den Aufbau der Prozesskette. Darüber hinaus wurde eine grafische Benutzeroberfläche erstellt, um das Definieren von Prozessketten über Python-Code in GTlab zu erleichtern. Die Benutzeroberfläche beinhaltet einen grafischen Editor mit automatischer Befehlsvervollständigung, die alle im Kontext verfügbaren Variablen, Methoden und Klassen vorschlägt. Die Befehlsvervollständigung unterstützt somit den Aufruf der vordefinierten Methoden zur Erstellung von Prozesselementen. Zudem wurde eine grafische Unterstützung zur Konfiguration von Prozesselementen implementiert. Dazu wurde der Aufruf der individuellen Benutzeroberflächen der Prozesselemente ermöglicht. Die darüber vorgenommenen Konfigurationen werden automatisiert in Python-Programmcode überführt und in den Editor eingefügt. Auf diese Weise kann ein Anwender die Prozesselemente über die gewohnten Benutzeroberflächen erstellen. Der Programmieraufwand zur Konfiguration von Prozesselementen wurde verringert.

Abschließend wurde die Funktionalität des Tasks in einem Anwendungsfall zur Massenabschätzung eines Axialverdichters demonstriert. Dabei wurde die gewonnene Flexibilität im Aufbau von Prozessketten verdeutlicht. Über Python-Skripte können zukünftig Prozesselemente in flexibler Reihenfolge, beliebig oft und in Abhängigkeit von zuvor berechneten Ergebnisdaten ausgeführt werden. Bei der Ausführung der Skripte können Daten aus dem zentralen Datenmodell ausgewertet und modifiziert werden. Zudem besteht die Möglichkeit neue Berechnungsverfahren über Python-Programmcode direkt in GTlab zu implementieren. Dabei muss keine Anpassung am GTlab-Programmcode vorgenommen werden, der das erneute Kompilieren einzelner Programmteile notwendig macht.

Die konventionelle Prozesssteuerung von GTlab ermöglicht das Erstellen von Pro-

zessketten bestehend aus Tasks und Prozesselementen. Tasks definieren standardisierte Ausführungsroutinen für Prozesselemente und können Prozessketten in mehrere Teilprozesse gliedern. In der skriptbasierte Prozesssteuerung sind Tasks grundsätzlich nicht notwendig, da sich die Prozesselemente über Python-Code in frei definierbaren Routinen ausführen lassen. Das hat jedoch zur Folge, dass häufig wiederkehrende Ausführungsroutinen in jeder Prozesskette erneut implementiert werden müssen. Zukünftig ist die Integration der Tasks in die skriptbasierte Prozessteuerung möglich, um die standardisierten Ausführungsroutinen über Python-Code nutzen zu können. Des Weiteren könnte das Erstellen von Datenmodellobjekten für die skriptbasierte Prozesssteuerung ermöglicht werden. Dadurch würde die Möglichkeit entstehen, Strukturen eines Datensatzes über Python-Code modifizieren zu können. So könnten Berechnungsverfahren implementiert werden, die den strukturellen Aufbau eines Triebwerks ermitteln und in Form von Datenmodellobjekten virtuell aufbauen.

Literaturverzeichnis

[1] The Qt Company Ltd.: *QObject Class*. http://doc.qt.io/qt-5/qobject.html#details, Abruf: 15.08.2019

[2] The Qt Company Ltd.: *QMetaObject Class*. https://doc.qt.io/archives/qt-4.8/qmetaobject.html, Abruf: 15.08.2019

[3] Blanchette, J. ; Summerfield, M.: *C++ GUI-Programmierung mit Qt 4*. Addison-Wesley Verlag. – ISBN 9783827324641. – (2007) S. 467

[4] The Qt Company Ltd.: *Multithreading Technologies in Qt*. https://doc.qt.io/qt-5/threads-technologies.html, Abruf: 16.08.2019

[5] Python Software Foundation: *Embedding Python in Another Applicationk*. https://docs.python.org/2/extending/embedding.html, Abruf: 30.08.2019

[6] Python Software Foundation: *The Python Tutorial*. https://docs.python.org/3/tutorial/index.html#tutorial-index, Abruf: 18.08.2019

[7] Lutz, M.: *Learning Python*. O'Reilly Media. – ISBN 978–0–596–15806–4. – (2009)

[8] Link, F.: *About PythonQt*. https://doc.qt.io/archives/qq/qq23-pythonqt.html, Abruf: 18.08.2019

[9] Rumbaugh, J. ; Jacobson, I. ; Booch, Grady: *The unified modeling language reference manual*. Addison-Wesley Verlag (Bd. 1). – ISBN 9780201309980. – (1999)

[10] Alhir, S.: *Learning UML*. O'Reilly Media, Inc.. – ISBN 9780596003449. – (2003)

[11] Loudon, K. ; Grimm, R.: *C++ kurz & gut*. O'Reilly Verlag. – ISBN 9783897212626. – (2012)

[12] Reitenbach, S. ; Krumme, A. ; Behrendt, T. ; Schnös, M. ; Schmidt, T. ; Hönig, S. ; Mischke, R. ; Mörland, E.: Design and Application of a

Multidisciplinary Predesign Process for Novel Engine Concepts. In: *Journal of Engineering for Gas Turbines and Power* Vol. 141, Nr. 1. – (Januar 2019) S. 2

[13] The Qt Company Ltd.: *QVariant Class*. https://doc.qt.io/qt-5/qvariant.html, Abruf: 09.09.2019

[14] Becker, R.-G. ; Reitenbach, S. ; Klein, C. ; Otten, T. ; Nauroz, M. ; Siggel, M.: An Integrated Method for Propulsion System Conceptual Design. In: *Journal of Engineering for Gas Turbines and Power* , Nr. ASME Paper No. GT2015-43251. – (Januar 2019) S. 2

[15] Eilebrecht, K. ; Starke, G.: *Patterns kompakt: Entwurfsmuster für effektive Softwareentwicklung*. Springer Berlin Heidelberg, 2018. – ISBN 9783662579367

[16] Python Software Foundation: *Built-in Functions*. https://docs.python.org/3/library/functions.html#dir, Abruf: 03.09.2019

[17] Nöthen, M.: *Integration eines Python-Interpreters innerhalb der Triebwerksvorentwurfsumgebung GTlab zur Steuerung des Programmablaufs*. – (April 2019)

[18] Transportation Federal Aviation Administration, U.S. D.: *Airplane Flying Handbook*. – (2004) S. 14-1

[19] Rolls-Royce: *The Jet Engine*. 1996. – ISBN 0902121235

[20] Tong, M. T. ; Halliwell, I. ; J., Ghosn L.: A computer code for gas turbine engine weight and disk life estimation. , Nr. ASME Paper No. GT2002-30500. – (Juni 2002)

A CalcWrapper-Klasse

```
class CalcWrapper:
    def __init__(self, calc)
        self._calc = calc

        for i in range(len(self._calc.findGtProperties())):
            prop = self._calc.findGtProperties()[i]
            if prop.ident():

                funcName = re.sub('[^A-Za-z0-9]+', '', prop.ident())
                tempLetter = funcName[0]
                funcName = funcName.replace(tempLetter, tempLetter.lower(), 1)

                def dynGetter(self = self, propName = prop.ident()):
                    return self._calc.propertyValue(propName)
                setattr(self, funcName, dynGetter)

                tempLetter = funcName[0]
                funcName = funcName.replace(tempLetter, tempLetter.upper(), 1)
                funcName = 'set' + funcName

                def dynSetter(val, self = self, propName = prop.ident()):
                    self._calc.setPropertyValue(propName, val)
                setattr(self, funcName, dynSetter)

        helpers = HelperFactory.connectedHelper(self._calc.__class__.__name__)
        for i in range(len(helpers)):
            funcName = 'create' + helpers[i]

            def dynCreateHelper(name, self = self, helperName = helpers[i]):
                helper = HelperFactory.newCalculatorHelper(helperName, name, self.
                    _calc)
                return HelperWrapper(helper)
            setattr(self, funcName, dynCreateHelper)

    def __getattr__(self, name):
        return getattr(self.__dict__['_calc'], name)
    def __setattr__(self, name, value):
        if name is ('_calc') or hasattr(self.__dict__['_calc'], name) is False:
            self.__dict__[name] = value
        else:
            setattr(self.__dict__['_calc'], name, value)
    def __dir__(self):
        return sorted(set(dir(type(self)) + list(self.__dict__.keys()) + dir(
            self._calc)))
```

Abbildung A.1
Implementierung der *CalcWrapper*-Klasse zur umhüllung von Prozesselementen im Python-Kontext

B HelperWrapper-Klasse

```
1  class HelperWrapper:
2      def __init__(self, helper):
3          self._helper = helper
4
5          helpers = HelperFactory.connectedHelper(self._helper.__class__.__name__)
6          for i in range(len(helpers)):
7              funcName = 'create' + helpers[i]
8              def dynCreateHelper(name, self = self, helperName = helpers[i]):
9                  helper = HelperFactory.newCalculatorHelper(helperName, name, self.
                       _helper)
10                 return HelperWrapper(helper)
11             setattr(self, funcName, dynCreateHelper)
12
13     def __getattr__(self, name):
14         return getattr(self.__dict__['_helper'], name)
15     def __setattr__(self, name, value):
16         if name in ('_helper') or name.startswith('createGt'):
17             self.__dict__[name] = value
18         else:
19             setattr(self.__dict__['_helper'], name, value)
20     def __dir__(self):
21         return sorted(set(dir(type(self)) + dir(self._helper)))
```

Abbildung B.1
Implementierung der *HelperWrapper*-Klasse zur umhüllung von Helferklassen im Python-Kontext